PhD Thesis

Improving Application Quality using Mobile Analytics

JULIAN HARTY
ORCID 0000-0003-4052-0054

Wednesday 31st May, 2023

School of Computing and Communications
Faculty of STEM
The Open University
“Consider it pure joy, my brothers and sisters, whenever you face trials of many kinds, because you know that the testing of your faith produces perseverance. Let perseverance finish its work so that you may be mature and complete, not lacking anything.” James 1:2-4

My family didn’t know what I was getting into a decade ago when I naively decided to start a part-time PhD while also working as a consultant in industry. They certainly became rudely aware in the last year of the process when I spent far too long wrestling with \LaTeX\ and with far far too much material written in gobs of words. Thank you so much for humouring this endeavour; let’s see where it leads!

My parents, and my mum in particular, would have been so pleased to learn I made it this far. Neither achieved an undergraduate degree even though both were very active and passionate about helping people through education. Seeing their children achieve brought them tremendous joy.

For anyone looking to follow my madcap idea of undertaking a part-time PhD degree as a mature student, I wish you all the best. I know of two friends currently on the journey, Isbael and Ego, and I really appreciate both your help and moral support. Good luck with your journey and endeavours. Note: many failures also shaped the research and the learnings through the PhD journey. I have at least as many abandoned and rejected papers as those which were accepted. Persistence is very necessary, as is the friendship and support of peers.
Abstract

The purpose of this research is to investigate and report on how mobile analytics can help real-world developers improve the quality of their apps efficiently and effectively. The research also considers the effects of mobile analytics in terms of the artefacts developed and maintained by the development team and also researches key characteristics of a range of mobile analytics tools and services.

Research Design: the research takes a developer-oriented perspective of using three complementary sources of data: 1) platform-level analytics, using Android Vitals as the primary analytics tool, 2) in-app analytics with a focus on runtime failures caused by crashes and freezes (known as Application Not Responding (ANR) in Android), and 3) interviews with developers. Action research techniques included roles of embedded developer, guide, and observer across different mobile app projects I was involved in. Hackathons were used to experiment with the speed and ability to find and address issues reported by the analytics tools used by the app developers. Their apps have a combined active user base of over 3,000,000 users. Many of these apps use a mainstream crash analytics library which was used to complement and contrast the results provided in the primary analytics tool. The research is intended to facilitate ease of future research and reproducibility, e.g. by using open-source projects as the code, bug reports, etc. are all published and available. This research was complemented by a) collaborating with professional developers who provided additional examples and results, and b) investigating grey material including grey literature and grey data.

The findings of this research highlights that using mobile analytics helped to reduce failure rates markedly, quickly, and effectively by applying techniques described here. Various limitations and flaws were found in the analytics tools; these provide cause for concern as they may affect the app’s placement in the app store and revenues. These limitations and flaws also make some issues in the apps harder to identify, prioritise, and fix. We identified ways to compensate for many of these and developed open-source software to facilitate additional analysis. Flaws and bugs were reported to the Android Vitals team at Google who acknowledged they would fix several of them. Several bugs were hard to reproduce, partly as Google deliberately hid pertinent details from the data they gather. Nonetheless app developers were able to ameliorate or fix the bugs for some issues even when they were not able to reproduce them.

Android Vitals shows the potential of how the combination of an app store and platform could be used to improve the quality of apps without users needing to actively participate. Some crashes were hard to reproduce and may be impractical to find before the app is released to end users. Developers can determine comparative improvements in their releases, such as whether they fixed a bug, by using Android Vitals and similar analytics tools; i.e. mobile analytics may help teams to determine whether they have improved the quality of their app even with flaws and limitations in the mobile analytics.
Acknowledgements

This research was inspired through my work with real-world developers of very successful mobile apps through the early and juvenile years of mobile app development (from 2006 to 2013) where I learned of the power and potential of using mobile analytics to improve the software and how teams create, test, and support that software.

Professor Arosha Bandara has been with me as my lead supervisor from the outset all the way through the journey, I simply wouldn’t have finished without him. Professor Marian Petre similarly has supported and encouraged me in my research and dedicated many hours to helping me get this thesis finally written. To Dr. Sheep Dalton who was my co-supervisor at the start and until I completed the probation report, he provided rich and fresh ideas and insights as I was shaping my research. Professor Yijun Yu kindly replaced Sheep when he moved to another university and always contributes fresh perspectives, insights, and ideas.

The many and various people involved in the app- and tool-centric case studies made this research viable and realistic, thank you for all your insights and contributions. And a final big thank you to Joseph Reeve who has led the development of most of the software created as part of this research.
1 Publications

My various contributions include:

1.1 Peer-reviewed papers


1.2 Private reports


1.3 Books


13. “The Mobile Developer’s Guide to the Galaxy” series of books, from 6th to 18th Editions. A widely distributed and read book that introduces the many and various challenges a developer of mobile apps needs to consider. The series is a collaborative project with multiple authors and editors where the book is published in print and electronically. The content is released under a creative commons license. I variously co-edited the book, wrote and revised various chapters both solely and with other authors. My main chapters include: Testing, Mobile Analytics, and Collecting and Understanding User Feedback. The current edition is freely available online at www.open-xchange.com/resources/mobile-developers-guide-to-the-galaxy/

1.4 Software

Please see the Software Contributions chapter in the appendices for details.
# Contents

Abstract v

Acknowledgements vii

1 Publications viii

1.1 Peer-reviewed papers viii

1.2 Private reports viii

1.3 Books viii

1.4 Software viii

Contents ix

Sourcecode Listings xvii

Preamble 1

1 Introduction 3

1.1 Background and Motivation 3

1.2 Research focus 5

1.3 Research Questions 8

1.4 Research contributions 10

1.4.1 Contributions to knowledge 10

1.4.2 Practical impact 11

1.5 Outline of this dissertation 12

2 A grounding in the domain 13

2.1 Mobile app ecosystem: apps and apps stores 13

2.1.1 Trust relationships 16

2.1.2 Communications paths and data flows 16

2.1.3 Membership criteria of each group 18

2.1.4 What participants can and cannot do (and who dictates the rules?) 19

2.1.5 Phases of a release 20

2.2 Mobile app software development practices 22

2.3 Mobile app development lifecycle 24

2.4 Mobile app usage lifecycle 25

2.5 DevOps for mobile apps 26

2.6 Software releases for mobile apps 28

2.7 Chapter summary 31

3 Related Work 33

3.1 Literature review method 34

3.2 Software Development 36

3.2.1 Software development practices 36

3.2.2 Software quality, focusing on reliability including measuring aspects 37

3.2.3 Stability and Reliability 38

3.2.4 Software analytics 39

3.2.5 Summary 42

3.3 App Stores and their Effects on Software Development and Engineering 42

3.3.1 Managing releases in app stores 46

3.3.2 Power dynamics 47

3.3.3 Summary 47
5.9.2 Contributions to opensource mobile analytics projects ........................................ 119
5.9.3 Android log-centric experiments ........................................................................ 120
5.10 Augmenting the app-centric case studies: sourcecode analysis ........................... 120
5.11 Tool-centric: Crashlytics ...................................................................................... 121
5.12 Tool-centric: Firebase Analytics ........................................................................... 122
5.13 Tool-centric: Google Play Console with Android Vitals ..................................... 123
5.14 Tool-centric: Iteratively with Amplitude .............................................................. 124
5.15 Tool-centric: Microsoft App Center ..................................................................... 125
5.16 Tool-centric: Sentry ............................................................................................. 126
5.17 Miscellaneous sources ......................................................................................... 127
5.18 Summary of the case studies overview ................................................................. 128

Findings 129

6 Findings: Analytics in Use 131
6.1 Motivating factors ................................................................................................ 132
  6.1.1 Successful apps ............................................................................................... 132
  6.1.2 Motivating devs to use mobile analytics ......................................................... 133
  6.1.3 Choosing mobile analytics .............................................................................. 134
  6.1.4 Ethics and Personally Identifiable Information (PII) ...................................... 136
6.2 SNAFU and real-world conditions ....................................................................... 137
  6.2.1 Engineering tradeoffs .................................................................................... 139
  6.2.2 Sense-making and decision-taking by developers ........................................... 140
  6.2.3 Sources of suboptimal practices ..................................................................... 142
6.3 Competence and optimisations ......................................................................... 142
  6.3.1 Using issue databases and preserving information .......................................... 143
  6.3.2 Engineering leadership .................................................................................. 143
  6.3.3 Triage ............................................................................................................. 144
  6.3.4 Hard to fix issues ............................................................................................ 144
  6.3.5 The fix process ............................................................................................... 144
  6.3.6 Integration of mobile analytics ...................................................................... 146
6.4 Benefits and achievements ............................................................................... 147
  6.4.1 Proactive fault finding .................................................................................... 147
  6.4.2 Bug reproduction ............................................................................................. 148
  6.4.3 Pre-launch testing ........................................................................................... 149
  6.4.4 Improving engineering practices .................................................................... 150
6.5 General topics ..................................................................................................... 150
  6.5.1 Scaling the use of analytics .......................................................................... 150
  6.5.2 Mobile analytics in a larger context ............................................................... 152
6.6 Improvements to the use of Mobile Analytics ..................................................... 153
6.7 Chapter summary ................................................................................................ 154

7 Findings: Apps and their Artefacts 157
7.1 Code-facing topics ............................................................................................... 158
  7.1.1 Build scripts ................................................................................................ 158
  7.1.2 Calls to the Mobile Analytics SDK ............................................................... 160
  7.1.3 Use of automated tests for the app ................................................................. 162
  7.1.4 Tests for mobile analytics SDKs ................................................................. 162
7.2 Bug-facing topics ................................................................................................ 163
### 7.3 App-facing topics

- **7.3.1 Improvements in crash rates**
- **7.3.2 Improvements in ANR rates**
- **7.3.3 Build scripts can affect the behaviour of the app**
- **7.3.4 User Experience of apps with in-app analytics**

### 7.4 Macro topics

- **7.4.1 Tradeoffs**

### 7.5 Improvements to the artefacts

- **7.5.1 Improving the source code**
- **7.5.2 In-app crash analytics**
- **7.5.3 Logging errors**
- **7.5.4 Using breadcrumbs**
- **7.5.5 Analytics for each release zone**
- **7.5.6 Preserving reports and results**

### 7.6 Chapter summary

### 8 Findings: Mobile Analytics Tools and their Artefacts

- **8.1 Design**
  - **8.1.1 SDK design**
  - **8.1.2 Developer experience**
- **8.2 Fitness-for-purpose**
  - **8.2.1 Product Fit**
  - **8.2.2 Actionable Reports**
  - **8.2.3 Integration into workflows**
  - **8.2.4 Programmatic access to analytics outputs**
  - **8.2.5 Android Vitals**
- **8.3 Utility**
  - **8.3.1 Efficacy of the tool**
  - **8.3.2 Benefits of combining tools**
  - **8.3.3 Pre-launch reports**
- **8.4 Dependability**
  - **8.4.1 Flaws in the mobile analytics tools and/or services**
  - **8.4.2 Link rot**
  - **8.4.3 Testability**
- **8.5 Cross-cutting topics**
  - **8.5.1 Fidelity**
  - **8.5.2 Ethical considerations**
- **8.6 Improvements to mobile analytics tools**
  - **8.6.1 Improvements to Google Play Console with Android Vitals**
  - **8.6.2 Improving integration**
  - **8.6.3 Improving the auditability and verifiability**
  - **8.6.4 Improving the analytics performed**
- **8.7 Chapter summary**

**Discussion, Conclusion, and Future Work**

### 9 Discussion, Conclusions, and Future Work

- **9.1 Discussion**
  - **9.1.1 Applying mobile analytics**
  - **9.1.2 Software development practice**
9.1.3 Reliability of mobile analytics tools ........................................ 231
9.2 Summary of contributions ...................................................... 233
9.3 Future Work ........................................................................... 236
  9.3.1 Investigation of additional app ecosystems ............................ 236
  9.3.2 Establishing transparency and trust in mobile analytics ........ 236
  9.3.3 Operational considerations when using Mobile Analytics ....... 237
  9.3.4 Combining human- and mobile- analytics feedback ............... 237
9.4 Final thoughts ......................................................................... 239

APPENDICES .............................................................................. 241
A Data analysis examples ............................................................. 243
  A.1 Kiwix experiment (at hackathon) ............................................ 243
  A.2 Crashes in cross-platform containers ..................................... 247
  A.3 Connecting failures pre- and post-app launch ....................... 250
  A.4 Google Play Console reports ............................................... 252
  A.5 Is Android Vitals Alive? ...................................................... 258
B Thematic Analysis .................................................................... 263
  B.1 Spreadsheet design and use .................................................. 264
    B.1.1 Worksheets ....................................................................... 264
    B.1.2 Compound names and full-stops ..................................... 264
    B.1.3 Spreadsheet “named ranges” ......................................... 264
    B.1.4 Spreadsheet formulae ................................................... 265
    B.1.5 Inclusion criteria for a findings chapter ......................... 266
  B.2 Ishikawa Diagrams ................................................................. 266
C Micro-experiments .................................................................... 271
  C.1 Adding meta-data to crash logs ............................................. 271
  C.2 Experiences with a test Android app: Travel Europe ............. 272
    C.2.1 From Zero to Ten ........................................................... 272
    C.2.2 Pre-launch report automated testing ............................... 273
    C.2.3 Google Play Console and Android Vitals reporting ........ 273
    C.2.4 Microsoft App Center reporting ................................... 273
    C.2.5 Conclusions ................................................................. 273
D Software Contributions ............................................................. 275
  D.1 Co-developed GitHub projects ............................................. 275
    D.1.1 Android Stability Analysis ............................................. 276
  D.2 Solo projects ......................................................................... 276
  D.3 Contributions to external projects ........................................ 277
  D.4 Software we developed for Google Play Console ................. 278
    D.4.1 Background ................................................................. 278
    D.4.2 Summary Data is available for download ....................... 279

Bibliography .............................................................................. 281
List of Terms .............................................................................. 303
Alphabetical Index ...................................................................... 313
# List of Figures

1.1 Sources of feedback for developers .......................................................... 7
1.2 Six Perspectives of Mobile Analytics ......................................................... 10

2.1 The Modern Mobile App Ecosystem: Android ............................................. 13
2.2 Four Views of an App Store ....................................................................... 14
2.3 Google Play message for Pocket Paint: To help better protect our users... 15
2.4 App Store: Communications Paths and Data Flows .................................. 17
2.5 Mapping life span stages for a mobile app release in an app store ............ 21
2.6 Dev Practices with Mobile Analytics .......................................................... 22
2.7 CI/CD development and feedback, reproduced from [42] for ease of comparison .. 24
2.8 Google Play App Development and Feedback .......................................... 25
2.9 Mobile App Usage Lifecycle ..................................................................... 25
2.10 Concept of applying feedback on software in DevOps ............................. 27
2.11 PocketPaint Active Users 180 days by app version ............................... 29
2.12 Kiwix Android Active Users 30 days by app version ............................. 31

3.1 Structure of this related work chapter ......................................................... 34
3.2 Overview of the literature review process and outcomes ........................ 35
3.3 Software Analytics, Buse and Zimmerman (2010) .................................. 39
3.4 Framework for presenting the state-of-the-art in software engineering for mobile apps, adapted from [79] ................................................................. 45
3.5 Various stakeholders in the modern app store ecosystem, adapted from [79] 45
3.6 XKCD’s views on App Store Ratings and Reviews .................................. 61

4.1 Methodology six perspectives (repeated) .................................................. 69
4.2 Categories of research methods .................................................................. 76
4.3 Boston matrix for app and mobile analytics tool ...................................... 79

6.1 Analytics-in-Use fishbone diagram ............................................................. 132
6.2 Sense-making process when development teams apply mobile analytics 141
6.3 Kiwix Version 2.5 OutOfMemoryError Exception ...................................... 149
6.4 Moonpig: User Review in Google Play ‘Always crashes...’ ....................... 152

7.1 Apps and their artefacts fishbone diagram ............................................... 158
7.2 Kiwix Android – 55 crashes for one user .................................................. 165
7.3 Fabric Crashlytics report on improvements in Pocket Code’s crash rate 167
7.4 Moonpig: Firebase Analytics illustrating two crashes that reached production 167
7.5 Kiwix crash rate drops once custom networking code replaced .............. 173

8.1 Analytics Tools and their Artefacts Fishbone Diagram .............................. 180
8.2 No Android Vitals reports on 16th March 2020 ..................................... 184
8.3 Alerts and graphs in Android Vitals on 26th March 2020 ....................... 185
8.4 Missing data reported in Sentry, in March 2020 ...................................... 186
8.5 Fabric Crashlytics Privacy Policy (in 2015) ............................................. 187
8.6 Iteratively: Product/Market fit ................................................................. 190
8.7 Addressing the contents of Mobile Analytics reports ............................ 193
8.8 Google Play Console Dashboard for PocketCode Lifetime User KPIs ....... 198
Sourcecode Listings

1. Install Sentry build.gradle to an Android app’s codebase .................................................. 158
2. Configure Sentry for that Android app .......................................................... 159
3. Writing code to verify the install and configuration of the Android app .......... 159

7.1 Privacy Policy for Pocket Code .......................................................... 169

4. Extract of false positive test failure for the Kiwix Arabic WikiMed app ......................... 205
5. Extract of pre-launch crash report A for Kiwix Android app ........................................ 206
6. Extract of pre-launch crash report B for Kiwix Android app ........................................ 206

7. AndroidX preference library example ........................................................ 230
8. One of the Kiwix Android crash fixes in v.2.5.3 .................................................. 244
9. Extract of java code that adds meta-data to crash stack traces to an Android app’s codebase ...... 272

D.1 Example of lines to match .......................................................... 276
List of Tables

3.1 Feedback sources about their app for developers ........................................... 59
4.1 Mapping data source (rows) to the 6 perspectives (columns) ............................ 73
4.2 Mapping research methods (rows) to the 6 perspectives (columns) ................. 77
4.3 App-centric cases: the research perspective ..................................................... 93
4.4 Tool-centric cases: the research perspective ..................................................... 94
5.1 Case Study key facts: GTAF ................................................................. 101
5.2 GTAF: data sources .............................................................................. 102
5.3 Case Study key facts: Local Halo ............................................................. 103
5.4 LocalHalo: data sources ...................................................................... 104
5.5 Case Study key facts: Moodspace .............................................................. 105
5.6 Moodspace: data sources ..................................................................... 106
5.7 Case Study key facts: Moonpig ................................................................. 107
5.8 Moonpig: data sources ........................................................................ 108
5.9 Case Study key facts: SmartNavi ............................................................... 109
5.10 SmartNavi: data sources ..................................................................... 110
5.11 Case Study key facts: Kiwix ................................................................. 112
5.12 Kiwix: data sources ......................................................................... 113
5.13 Case Study key facts: Catrobat ............................................................... 114
5.14 Catrobat: data sources ..................................................................... 115
5.15 Case Study key facts: CI .................................................................... 117
5.16 Tool Centric Case Study key facts: Crashlytics ............................................ 121
5.17 Tool Centric Case Study key facts: Firebase Analytics ................................. 122
5.18 Tool Centric Case Study key facts: Google Play Console with Android Vitals ... 123
5.19 Tool Centric Case Study key facts: Iteratively with Amplitude ....................... 125
5.20 Tool Centric Case Study key facts: Microsoft App Center ............................ 126
5.21 Tool Centric Case Study key facts: Sentry ............................................... 127
7.1 Android Vitals: Improvement in crash-rate for Pocket Code post WebView improvements 166
7.2 Android Vitals: Improvement in crash-rate post Hackathon ......................... 167
7.3 Improvements in Kiwix Crash Rates for the main Kiwix app and WikiMed in English 171
7.4 Pocket Code: ANR rate for Last 7 days, as recorded on 1st December 2019 .......... 171
8.1 Flaws discovered in Google Play Console with Android Vitals .......................... 208
Preamble
Introduction

1.1 Background and Motivation

Before this research, little was known about developers’ integration of mobile analytics into their artefacts and/or their processes. Unknown were how, why, and when they used the outputs of the mobile analytics. Similarly, the effects of using mobile analytics in terms of any changes to the reliability of these apps was not known. Furthermore, little was known about the mobile analytics tools and services used by app developers in industry or in real-world mobile apps.

Achieving quality in software is challenging: software is not perfect and no developer can create perfect software. This section elaborates on some of the challenges of delivering high quality software for mobile devices.
and the characteristics of the mobile app ecosystem that make reliability a particularly important quality factor.

**Software is not perfect:** it is formed through numerous human endeavours using flawed tools and techniques. Bugs are ubiquitous in software, and no practical software is bug-free. Even one of the most respected software engineers, Donald E. Knuth, recognises, publicly acknowledges, and pays for, bugs found in his creations. “You are entitled to a reward of at least 0x(2.56) if you are the first person to report a bona-fide error not on those lists.” [13]. And self-aware developers expect that there will be bugs in their software. Nonetheless those involved can improve software through their choices and practices.

**Nobody is perfect:** even leading technology companies release software that crashes. Some have the misfortune to release software that crashes many other apps inadvertently, as Google discovered in Spring 2021 [14]. App developers, including the BBC for their iPlayer app, posted advice to end users about the issue in Google Play [15], as the adverse effects of the bugs introduced by Google were so widespread.

As noted above, quality is important for mobile app developers because of the widespread use and impact of their software. Quality factors like app reliability are particularly important because of the way the mobile app ecosystem works. This is because of the developers’ dependence on the providers of mobile app platforms for the distribution of their apps and how these providers use reliability measures to include, promote, and sometimes exclude apps from the ecosystem.

**Mobile app ecosystem:** to survive, app developers need to deliver software that can be used successfully by end users. To do so they need to be able to create software, package it as an app, and distribute it so it is available to end users. They need to do so in a timely manner - perfection may be too long to wait for! There are significant competitive pressures [16, p. 117]. Therefore developers need to balance and make tradeoffs in the work they do and don’t do. As Tantithamthavorn wrote: “Bad decisions cost money (and reputation) so we need better tools for making better decisions.” [17, p.115]. The article also observes developers also need to decide what to avoid doing. Therefore developers need help to decide which failures are appropriate to fix now (and which to leave be).

End users need to be able to install and use the app. If the app is sufficiently usable and useful and behaves adequately, they may continue to use it. Developers cannot assess *a priori* whether their app will meet the needs and expectations of users, or the needs of their stakeholders. However, they can increase the likelihood of users continuing to use their app if they can ensure the stability of the app by preventing the app from crashing or becoming non-responsive.

**Stability:** is a prerequisite for an app to be viable in the marketplace, and several key app stores clearly state that unreliable apps (e.g. apps that crash) will be marked down and possibly ejected from the app store [18–20]. For example, Apple states “On average, over 40% of app rejections are for Guideline 2.1 – Performance: App Completeness.” [21] with

---

1: Google uses the term stability as their measure of reliability of Android apps in particular so both appear in this dissertation, details follow in Section 1.3
crashes listed as the first rejection reason.

Additionally, a survey in Germany reported that the most important annoyance for users was instability, *i.e.* “instability (app crashes at startup or in certain cases) with 41% of all responses” [22]. Crashes are undesirable yet commonplace; therefore they are a good candidate for evaluating how analytics may be able to a) help measure them and b) help developers to address them, in order to improve the reliability of mobile apps for end users.

Conversely, Android apps that score highly in terms of stability may be selected to be promoted and featured in Google Play. One of the developer-oriented Google Play Guides provides an imperative: ‘Improve your app’s quality and discoverability’ [23] which combines with the Google Play Guide on Android Vitals. This states: “Apps whose metrics are higher have greater promotability, which raises their ranking in Google Play Store searches. They also are more likely to be eligible for the New & Updated and Editor’s Choice collections on Google Play, and to be nominated in the Google Play Awards.” [24].

Crashes can leave indelible, adverse, results. An increase in crashes led to an increase in ‘churn’ [2] of up to six times the average rate of churn according to industry reports [25, 26] [3]. Facebook deliberately tested the tolerance of users by introducing automatic crashes in their Android app, according to [27], to measure when users would abandon the app entirely.

In terms of bugs mobile developers face, “... automatic in-app crash reporting is the most prolific channel of reporting bugs... ” [28] – presumably as there are many crashes in mobile apps, otherwise it wouldn’t be a prolific source.

Software Analytics: exist and are used across and throughout software development practices, and they can be used to better understand and improve these practices and the resultant software [29–31]. A major source of software analytics is generated through use of the software, *i.e.* use of mobile apps in the context of this research.

Analytics can be especially important for improving mobile app reliability because there is a huge diversity in devices and contexts where mobile apps are used, which can affect reliability of the apps. However, we have limited understanding of how software developers apply mobile analytics in practice for improving app reliability. To address this gap, we study the use and utility of mobile analytics by multiple Android mobile app development projects, the largest and most ubiquitous mobile app ecosystem.

1.2 Research focus

These observations regarding the relationship between app reliability and success in the ecosystem highlight the importance of understanding how developers can improve their ability to address this aspect of app quality. This leads to the primary motivation for this research, which

[22]: Nitze et al. (2015), ‘A survey on mobile users’ software quality perceptions and expectations’

[23]: Android Developers (2020), Improve your app’s quality and discoverability

[24]: Android Developers (2020), Use Android vitals to improve your app’s performance, stability, and size

2: Churn is also known as attrition and is used to measure users who leave a collective group over a specific period en.wikipedia.org/wiki/Churn_rate.


[26]: Levy (2017), The “Crash and Burn” Report Findings

3: Note: Apteligent was acquired by VMWare in 2017 and few of their online materials remain available.

[27]: Efrati (2016), Facebook’s Android Contingency Planning

[28]: AlSubaihin et al. (2019), ‘App store effects on software engineering practices’

[29]: Buse et al. (2010), ‘Analytics for Software Development’

[30]: Buse et al. (2012), ‘Information needs for software development analytics’

[31]: Menzies (2018), ‘The Unreasonable Effectiveness of Software Analytics’
aims to understand how software analytics can help improve quality, particularly with respect to reliability by gathering and analysing data relating to failures during real-world use.

**The Big Picture:** broadly, this research aims to understand whether mobile analytics can help developers to improve the reliability of their apps. A second applied research area emerged as various flaws and limitations were discovered in the various mobile analytics tools and services, to discover and categorise the flaws and limitations, and to consider some of the effects on actually improving app quality given these flaws and limitations.

This research concentrates on platform-level analytics together with crash and error analytics collected by in-app analytics. It applies these forms of analytics with a focus on improving stability of Android apps available in the Google Play Store. The focus on Google Android is based on identifying the most used operating system – Android – in its most popular and mature platform – Google Android – to discover whether mobile analytics can help developers improve the stability of their apps by fixing bugs that adversely affect their stability. Further, the research focuses on apps that can be installed and run on smartphone-like devices, including tablet devices.

---

### Usage data

Usage data can be generated by actual users, emulated by other people, simulated by automated scripts interacting with apps, generated by programs, and fabricated by providers of mobile analytics tools/services. Of these, usage generated by actual users is inherently part of the real-world microcosm developers of the apps inhabit and therefore the research methods need to understand the use of mobile analytics in this context.

There may be various ways to access the real-world data, such as being a part of the development team, or being granted access by the development team. While the providers of the mobile analytics services may also have access various ethical, commercial, non-disclosure, safeguarding, and additional considerations make such access unlikely even though it could be tremendously rich and insightful. Some of the research can be performed using emulators and/or simulators and allows for greater control of the input conditions.

---

From a research perspective focusing on the **Android** platform is also credible in terms of specialising the research focus given its mainly opensource codebase and popularity as a platform (Number 1 globally). Android is also well researched, for example with 40 papers on Android (of these 3 also included iOS), whereas only 2 papers were exclusively focused on the extremely popular iOS platform at the preeminent International Conference on Software Engineering (ICSE) 2022 conference and related workshops. Android, therefore, has much richer range of prior-art to build on; and any improvements in mobile apps via this research has the potential to effect improvements in the largest of the mobile ecosystems.
Developers have various sources of feedback about their apps, as Figure 1.1 illustrates. The pink triangle represents the extent of Google Play (the app store) in terms of providing feedback. Other feedback is also available independently of the app store, for instance: from the development process, and by using software incorporated directly into the app that provides bi-directional communications between the development team and end users.

Each source of feedback may stem from humans (for example, in reviews) or from software (for example, from code quality tools such as Lint). The analytics data used in this research draws on three sources of software-generated feedback:

1. Pre-launch testing: automated checks and testing provided by the app store (Google Play) as part of their Pre-Launch Report (PLR),
2. platform-provided analytics: automated analytics and reports provided by gathered Android Vitals, which is integrated into the Google Android platform and available free of charge for all developers of apps in the Google Play app store,
3. in-app error reporting (including crash reporting): software added by the app development team to detect and report crashes, and optionally errors and similar/related data. In app crash reporting is used in at least 80% of Android apps in Google Play according to AppBrain’s analysis of Crash Reporting libraries.

Figure 1.1: Sources of feedback for developers
The Research focuses on automated checks and tests performed by the app store, automated analytics and reports both in-app and at the platform level. Other sources of feedback available to developers includes feedback from people and feedback from their development tools and CI automation.

Developers have various sources of feedback about their apps, as Figure 1.1 illustrates. The pink triangle represents the extent of Google Play (the app store) in terms of providing feedback. Other feedback is also available independently of the app store, for instance: from the development process, and by using software incorporated directly into the app that provides bi-directional communications between the development team and end users.

Each source of feedback may stem from humans (for example, in reviews) or from software (for example, from code quality tools such as Lint). The analytics data used in this research draws on three sources of software-generated feedback:

1. Pre-launch testing: automated checks and testing provided by the app store (Google Play) as part of their Pre-Launch Report (PLR),
2. platform-provided analytics: automated analytics and reports provided by gathered Android Vitals, which is integrated into the Google Android platform and available free of charge for all developers of apps in the Google Play app store,
3. in-app error reporting (including crash reporting): software added by the app development team to detect and report crashes, and optionally errors and similar/related data. In app crash reporting is used in at least 80% of Android apps in Google Play according to AppBrain’s analysis of Crash Reporting libraries.

5: The image was created in pikochart, a paid for online editing tool. Thanks to Silvia Harty for her help designing the original figure.

6: Android Vitals is integrated into Google Play Console which provides additional functionality including app store metrics, ratings and reviews, release management, and so on.

7: https://www.appbrain.com/stats/libraries/tag/crash-reporting/android-crash-reporting-libraries
In summary, the rationale for selecting Google Play Console with Android Vitals include the vast scale and reach of the platform-level analytics in the largest app store globally where potentially billions of users rely on the quality of these apps and millions of developers have to trust and rely decisions that the Google Play store ecosystem applies for the apps they release in the app store. Furthermore the majority of Android apps include at least one analytics library that collects and reports crash and error analytics. Both users and developers de-facto trust these libraries are well behaved and can be depended on. Similarly users implicitly trust developers not to do net harm.

Case studies were used to study the effect of mobile analytics as they provide real-world, in-situ, microcosms of mobile analytics being used for apps that have external user-bases (rather than research based on students, friends and family, and so on).

The next section introduces the research questions explored in this research and outlines the contributions made by this work.

1.3 Research Questions

The research hypothesis is that using mobile analytics can help improve the work development of teams and the quality of the products they create. Here work includes the development, testing, and bug investigation of the software being created. For the quality of the product, this research is focusing on a subset of qualities which are Technology-facing, and in particular the reliability and stability of the mobile apps when they are in use.

This research builds on prior work on software analytics for mobile apps and focuses on its practical application by software development teams to improve app reliability. This is important because app reliability, as measured by crash rates and App Not Responding faults (ANRs), is a key metric in the mobile app ecosystem. Because this metric is used by the platform provider to determine whether an app should be given access to the ecosystem, it is something developers should pay attention to.

In order to understand the effect of applying mobile analytics on the software development practices for improving reliability of apps, we need to explore the processes, artefacts and tools associated with these activities. This research investigates these three dimensions based on case studies of different mobile app projects, to identify both the current practices and opportunities for improvement provided by using mobile analytics.

The core question investigated by this research:

*How can applying mobile analytics in software development practice improve the reliability of mobile apps?*

Expanding on key elements of the research question:

**Applying mobile analytics**: is the use of mobile analytics in order to effect improvements to the practices and the artefacts. Applying mobile analytics refers to both collecting data from the usage of the app and also making use of the analysis of this data to identify and
address issues that can improve the app. Improvement of the app focuses on increasing the stability/reliability by reducing ANRs, crashes, and through improving how the app handles various errors (typically reported through Exceptions).

**Mobile analytics:** Analytics where the data is collected by software running on mobile smartphone-based devices pertaining to the app’s qualities-in-use. This research focused on analytics collected pertaining to the stability of the app, where stability includes the reliability of the app.

**Software development:** includes tasks performed by the software developers including design, coding, testing, bug reporting, and bug tracking. Use of Scrum development practices, following recommendations and guides that include application compatibility, User Interface (UI) guidelines, and designing for performance and responsiveness, etc. [Software testing [33, pp.398 - 399].

**Reliability and Stability** are two intertwined measures of software quality in use. There are contradictory opinions on their relationships to each other and to their contributions to software quality. For the purposes of this research reliability is used whenever software engineering is in focus and stability matches Google’s use of the word for Android apps. These two terms will be discussed in Section 3.2.3. Stability and Reliability.

**In practice:** the key scope of measurement focuses on the efficacy in real-world projects from the perspective of software practitioners who develop mobile applications.

In order to answer this research question it is appropriate to consider improvements to the app (i.e. the product) and to the processes/practices development teams apply when they develop and maintain their mobile apps. Improvements cannot be be considered usefully in isolation; they need to be grounded in the current practices: the developers will have their perspectives on their use of mobile analytics, and their development artefacts may provide cross-verification of what they say they do compared with tangible evidence of how they use those mobile analytics tools.

Furthermore, there may be constraints and/or limitations in the current mobile analytics tools which may adversely affect the improvements development teams are able to make to their processes/practices and to their products. Hence, it is also germane to consider improvements to the current mobile analytics tools.

These lines of enquiry are relevant to the core research question and lead to six more granular questions each focusing on a distinct, yet related, perspective on the use and effect of mobile analytics.

The six perspectives are illustrated in Figure 1.2 and paraphrased below:

1a What do app developers say they do in terms of using mobile analytics? (understand the status quo from their perspective).

2a What’s possible in terms of improving their processes, their practices through using mobile analytics?)
1.4 Research contributions

There are two areas of contributions: to knowledge, and in terms of practical impact.

1.4.1 Contributions to knowledge

The research contributes to the understanding of tools and information seldom available to research – of professional app developers, their artefacts, and of professional mobile analytics tools and services.
Processes: This research contributes knowledge on the approaches various app development teams apply when they use mobile analytics including the selection, integration of code and services, and their application of mobile analytics to detect, identify, and address errors and failures reported by mobile analytics. It builds on prior research, for example, on Insight, and confirms their findings. It contributes new knowledge in the adoption platform-level and commercial in-app mobile analytics, including a) usage patterns by development teams ranging from individual developers, small teams and large, sharded teams, and b) public opensource projects, hybrid projects that combined private and proprietary practices, through to a development team at a major corporation.

Some of the findings were surprising in terms of the patterns of use and in the efficacy of using mobile analytics to achieve significant improvements.

Development teams who embedded mobile analytics into their ongoing, core practices, were able to achieve highly reliable and stable apps.

Artefacts: The research extended prior art in studies of opensource mobile app codebases, with a focus on the use of the most popular product offering: Firebase Analytics. Developers often incorporated multiple mobile analytics libraries into a single mobile app, each for a specific purpose. When developers addressed failures reported by mobile analytics they often modified the source code of the app; these modifications were generally small, yet the effects on improving the reliability/stability of the app were material.

It also contributes insights from proprietary, commercial codebases and issue tracking artefacts where development teams intermittently filed bug reports for issues reported by mobile analytics.

Mobile Analytics Tools: The research identified characteristics of a wide range of mobile analytics tools that serve Android app developers in particular. It also found and presents a range of flaws found in professionally-developed mobile analytics tools, including several of the most-used mobile analytics offerings.

The research contributes material relevant to professional app developers and to the developers of mobile analytics.

Improvements were identified in all three areas and some of these were implemented during the research.

Note: In addition to specifying my research contributions add explicit summaries of my contributions that have already been published.

1.4.2 Practical impact

Several of the tool development organisations including Amplitude, Google, and Iteratively actively sought insights and updates from my research. They improved various aspects of their respective mobile analytics offering.
App developers who applied the techniques described in my research were consistently able to significantly improve the reliability/stability of their apps.

1.5 Outline of this dissertation

At a high-level dissertation is in three main parts: the preamble which includes chapters 1 to 5, the findings in chapters 6 to 8, followed by the discussion, conclusion, and future work in chapters 9 and 10. There are two appendices: on thematic analysis, and additional details for some of the mini-experiments.

Chapter 2 | Preparing the ground: this chapter prepares the ground for the rest of this dissertation by explaining contemporary development practices for mobile apps. It then presents five conceptual models related to mobile apps. These are followed by several, relevant, practical details.

Chapter 3 | Related work: explores the state of the art relevant to software quality, software analytics, and the mobile app ecosystem.

Chapter 4 | Methodology: sets out the research approach adopted to gather and analyse data for the different case studies explored as part of the research.

Chapter 5 | Overview of the case studies: introduces each of the app-centric and tool-centric case studies using a consistent structure to make them easy to comprehend and to facilitate comparison.

Chapter 6 | Findings - Analytics in use: presents the key findings from the case studies relevant to the use of analytics by mobile app development teams.

Chapter 7 | Findings - Apps and their artefacts: discusses the findings relating to the software developed in the different mobile app case studies, together with their associated artefacts.

Chapter 8 | Findings - Mobile analytics tools and their artefacts: focuses on the mobile analytics tools explored during the course of the research and the different artefacts produced by these tools.

Chapter 9 | Discussion: explores the findings across the different perspectives of the research in relation to the wider literature of software quality and mobile app development practices.

Chapter 10 | Conclusions and future work: summarises the key contributions of the research and discusses avenues for further investigation.
As mentioned in the introduction, the mobile ecosystems touch on billions of people’s lives, where flaws in the apps and the ecosystem can adversely affect the lives of many of those people. Figure 2.1 illustrates the main parties engaged in the ecosystem, and it sets the context for the thesis and this chapter together with the research questions. The marketplace - the app store - establishes the ecosystem which generates revenues from the user population and frequently to a lesser extent from the makers - the developers of the apps. Various providers of utilities and services offer these to the makers and they may also obtain revenues from one or more parties that are part of the ecosystem and/or from others, including advertisers. In the context of this research, utilities include software libraries such as the opensource logging service, Timber, where developers are responsible for any service provision.

This chapter provides some background about the software development practices used for mobile apps, including the mobile app development lifecycle and related software releases. To put these practice in context, we start by describing the mobile app ecosystem.

### 2.1 Mobile app ecosystem: apps and apps stores

This section describes the ecosystem of mobile apps through an exploration of the characteristics of apps and app stores. It presents four views of apps in an app store together with various implications of the views, relationships and interactions.

The vast majority of mobile apps are provided through app stores. The two largest app stores: Google Play and Apple’s App Store. Their respective platforms both collect mobile analytics from end user’s devices with permission. So, understanding the conceptual model of apps and

---

1. inspired by Figure 1, on p. 96 of ‘A socio-technical view of platform ecosystems: Systematic review and research agenda’

2. github.com/JakeWharton/timber

3. Albeit the permission might be granted by default without the users making a conscious choice to do so.
app stores provides some context for the sources of mobile analytics.

The concept of an app store has existed since at least 2003, according to the co-founder and CEO of Salesforce [36], where the idea was proposed by Steve Jobs and later implemented as AppExchange in the Salesforce platform. Around the same period various app stores emerged for mobile apps; and the concept seems to have been introduced around 1999 by Handandgo. Academic research into the effects of app stores emerged in or around 2010, for instance with the work of Kimbler who investigated the effects on mobile operators from a business strategy perspective [37]. (Mobile operators lost out in the overall battle of app stores, now platform specific app stores dominate the market.)

The research is situated in apps that are available in app stores and in the Google Play app store specifically. App stores house millions of apps and serve billions of users. They also present a rich tapestry of perspectives on software apps and the ecosystem. There has been a great deal of research that focus on particular areas of these apps and sometimes connect these areas as part of the research. This research focuses on an area seldom investigated, namely it concentrates on the developer’s view of how their app is perceived by the app store and whether they can improve the perception by addressing sources of failures.

Figure 2.2 illustrates the four views; broadly, those closer to the centre can also see what those in outer rings can see. As a wise supervisor commented: “It’s a bit like standing at different elevations on a mountainside and looking out over the landscape - the higher you are, the more you can see”.

The first view is the public view of the app store, what is visible to someone who is not actively engaged with the app store. Examples include people who are not logged into their account, search engines, researchers mining the app store for ratings and reviews, and so on. The public is able to see aggregate ratings and some recent reviews for specific apps. Older reviews are generally hidden from public view (which may limit some research and search engine insights).
The next view is that of a user of a particular app or set of apps. They may have installed some of the apps directly, they are likely to also have pre-installed apps on their device too. They have the ability to interact with the app store, for instance they can see, create, and update their ratings and reviews. They can also see the public view.

Developers have the next view, which includes information the app store records about the developer’s interactions with the app store, and information the app store provides the developers directly (i.e. generated by the app store and related entities). The information also includes feedback provided by users via the app store (e.g. ratings and reviews). Developers can also see the public view, although they cannot see the entire view of their user-base. However they can see any rating and reviews provided by the users.

Authors and developers are the two end points of ratings and reviews. Authors create them and developers receive them and can choose to respond to them, at least in some app stores. Importantly, their primary communications goes via the app store, rather than being direct, and aspects of these communications are often public for a period. Authors and developers can see their individual ratings and reviews for much longer periods than presented in the public view. The app store can use the ratings to decide on the quality of the app and their assessment may affect various important facets of the app’s existence in the app store. For instance, well rated apps may be promoted and poorly rated apps may be demoted in search results. As Google states “Apps whose metrics are higher have greater promotability, which raises their ranking in Google Play Store searches.” Also, poorly rated apps are sometimes subject to additional scrutiny and delays in the release process, as illustrated in Figure 2.3 when the overall rating for a release dropped sufficiently to trigger this change.

The final view is that of the app store, the ‘storeholder’ in the figure. They have a global and holistic view of the entire store, including all the reviews, user interactions, and whatever usage activities have been performed by all the other three views.

We now cover various implications of this conceptual model of the app store.

---

6: If supported by the app store, for instance Google Play does.

7: A caveat on the use of potentially: this is because the app stores are closed systems with limited information about their actual behaviour in the public domain.
2.1.1 Trust relationships

One of the key success factors of the modern app store (typified by Apple’s App Store and Google Play) is that the platform provider provided the entire ecosystem and established the rules of engagement. The locus of trust is the provider of the app store, which acts as the public face and to some extent also acts as a representative for both the users and the developers. In terms of financial transactions it also acts as the intermediary and facilitates users being able to obtain refunds for paid-for apps and in-app purchases subject to various conditions.

Note: There are many details related to the trust relationships for those interested in that topic, however in the interests of focus and concision they are outside the scope of this thesis.

2.1.2 Communications paths and data flows

There are numerous communication paths for mobile apps both with and without an app store being involved. As the vast majority of apps and users use devices and apps that are part of an app store ecosystem (even if they are obtained from other sources, e.g., as often occurs in India). I will only consider the ecosystem that includes an app store in this dissertation.

The information about mobile apps can come from users directly or indirectly, directly if an app collects information or indirectly, from devices, where the data is collected by the platform. The collection of indirect data could be via Application Programming Interfaces (APIs) in the operating system, installed apps with privileges to access information about other apps, from accessibility services, and potentially other means e.g. installed viruses. Alternatively, it could come from intermediaries - particularly the app store, and also from network traffic, observers, etc.

Source code and source code repositories are also useful sources of information about mobile apps. Information can be usefully combined from several sources, for instance from source code about calls to write log messages compared to actual logs recorded when the app has been used on a device. Given the app store plays a pivotal role let’s consider its role in terms of communication paths as illustrated in Figure 2.4.

An app store is more than the store front, it controls and affects many aspects of the ecosystem that gathers around it. It is also more than the software, data, and information that the various memberships can access. For instance the modern app stores often include software that is mandatory and pre-installed on end-user devices where that software cannot be easily removed or disabled by users. This software includes a local storefront that offers end-users new apps, updates, and enables users to manage optional apps.

The app store provides various primary communications paths between the various parties involved in the ecosystem. It may be an active party, for instance in some of the reports provided to developers and/or users, and in policy-related matters; or it manages communications between app users and developers. Often the app store’s owners define the rules
of communications, including details such as whether and when apps can ask users to rate an app.

Some of the communications involves humans, or software chatbots masquerading as pseudo-humans intended to behave similarly to how humans would do in similar circumstances, for instance to provide in-app assistance [39] and to help developers respond automatically to app reviews [40]. Other communications is generated by software, for instance usage and diagnostic data collected by the operating system and related utilities on a mobile device (collectively described as the platform). This usage and diagnostic data in the Google Android ecosystem provides the source of the platform-level analytics.

As shown in Figure 2.4, there are two forms of data flows: explicit and implicit. Explicit data flows are actively and intentionally performed by one or more of the participants, implicit data flows represents information that can be inferred or gleaned from various actions and inactions. They are both feedback mechanisms for app developers.

Examples of actions intended to communicate explicitly include:

- Making the app available in the app store; this includes creating screenshots, a description of the app, adding meta data the app store requires and/or requests, etc. This information becomes public if the app store approves the app for release.
- Ratings and reviews performed by app users. Only a subset of users provide these, the percentage varies from zero to a maximum of

---

[39]: Baez et al. (2021), 'Chatbot integration in few patterns'
[40]: Greenheld et al. (2018), 'Automating Developers' Responses to App Reviews'
around 10% with typical percentages around 1% to 3%. Estimates vary, partly as the definitions vary too. AppBrain states 46.5% of Android apps do not have a rating. In comparison, 42matters.com estimate 41% of Android apps and 57% of iOS apps have no rating.

- Responses to reviews, for example Google Play allows developers to respond to reviews, and for both reviewers and developers to update their reviews and responses.
- Suspending an app so it is no longer available to users to download. Storeholders sometimes suspend apps and even developer accounts where they perceive the app and possibly the developer contravenes the app store’s policy.

Implicit information flows include:

- New releases of apps and related content (such as in-app content, often purchased using in-app purchasing). These indicate the developer wishes to actively engage their userbase. Upgrades may include changes to the app seeded by various sources such as ratings and reviews and other data, including:
- Usage data and upgrades, both imply the software provides some value to the users. Lack of usage may also be an indication the software is not currently providing value - this may be expected for instance with seasonal apps. Uninstalls are a stronger signal that users no longer see sufficient value in the app to keep it on their device.

On-device bug reports may be a hybrid, where the bug reporting utility on the device does much of the data collection and may report this automatically and transparently, however it may sometimes ask the user for additional input and permission to send the bug report.

### 2.1.3 Membership criteria of each group

As Figure 2.4 illustrates there are four numbered groups in the illustration. People can potentially belong to more than one group (albeit membership of the storeholders is limited to owners and those they assign membership to, e.g., as administrators of the app store). Group membership constrains what the members can do as participants and what they have access to.

1. **Public**: the membership criteria are minimal. Here ‘public’ is any entity, human or technological, that has access to the app store. An example of a technological entity, is a search engine crawler or software including web scraper technology and scripts that use APIs provided to obtain information about apps in the app store.

2. **App user**: the public can use an existing account or create a new account with the app store that would allow them to become an app user. Note: there may be restrictions or constraints that mean not everyone can install every app on every device, however the general practice is that apps are freely available for app store users to install on any device they possess.

3. **Developer**: developers need to be registered and validated by the app store, the process varies for specific app stores, they often involve payment of a fee and some amount of validating their
2.1 Mobile app ecosystem: apps and apps stores

A FIFTH MEMBERSHIP GROUP - PROVIDERS: we have already identified four membership groups involved in app store ecosystems, there is at least one more group, and also additional data flows in the ecosystem. The fifth membership group are service and/or utility providers; they were introduced in Figure 2.1. They are optional members in that they are selected by app developers who wish to use their utilities and/or services; they are not direct participants in the app store and their data flows primarily flow between the app developers and the provider.

They provide non-trivial functionality such as device farms for automated testing and other capabilities such as in-app analytics, feedback, and so on. Developers can choose to incorporate software libraries into their apps and use the services provided, for instance as conduits of communications between the app and the developers. Here developers include other specialist groups in their organisation such as customer service personnel and marketing teams. Many app developers choose to use at least one such service, some incorporate several and there is even specialist software that enables developers to manage multiple similar services within their apps on end-user devices. An example of this type of software is github.com/segmentio/analytics-android (other platforms are also supported and there are other providers of similar software). Additional data flows include those provided through using in-app analytics (implicit data) and in-app feedback (explicit data).

Mobile analytics is the most common choice provided to app developers as inferred from market statistics provided by AppBrain. Note: The final group in Figure 2.1 are device providers. As device providers are seldom directly involved with mobile analytics they are not included here.

2.1.4 What participants can and cannot do (and who dictates the rules?)

Here five group of participants are considered in terms of what they can - and cannot - do.

- Public: The public cannot review an app or easily download the app (so they do not provide data about apps either explicitly or implicitly). They can view publicly accessible information, including information that was gathered previously, potentially by others.
- App user: They can rate and review apps they have installed on their account or device, therefore they can provide data explicitly. They can also install, update and deinstall apps.
- Developer: Approved developers can upload apps to the app store and publish them if the app store also approves the release. They can choose to submit new versions of their apps, sometimes they may be required to do so by the app store. They can choose to provide implicit feedback.
suspend or withdraw their app from the store, note: generally users can continue to use the app if they have it installed. Developers are expected to interact with the app store and often do so of their own volition, for instance to see how their app is ‘doing’. The developer may define a price for their app and/or any in-app purchases. They may also require users comply with additional terms of use, and many apps do so.

- **Storeholder**: They are by far the most powerful participant as they establish the ecosystem including the rules of engagement and enforce these rules. The app store has the right of delay or veto of releases, it can suspend apps and developers, and much else besides. They are expected to comply with the laws of the various countries the app store is available in and also where their business is situated. These laws may affect the developers and the app users, for instance the amount of sales tax charged on a purchase in the app store.

- **Providers**: the fifth membership group are *service and/or utility providers*. These providers choose their rules of engagement app developers would need to abide by. App development teams choose whether to use them, and if so whether to pay for the services or limit themselves to free products and/or tiers of service. Power relationships tend to be weak as there is generally healthy competition and lots of offerings available to app developers - they can ‘vote with their feet’.

**Membership matters**: in particular because of who has access to which data and for how long they have access. Note: Control and ‘ownership’ of the data are also relevant topics, however they are not necessary to comprehend the rest of this topic. Chapter 9 in Section 9.3.2 proposes future work to research these topics.

### 2.1.5 Phases of a release

For any given release of a mobile app there are at least three material phases in order for the release to be used:

1. **Building the product**: which may incorporate practices and tools intended to ship a ‘quality product’. Some teams also incorporate logging and reporting to help measure the behaviours of the app in use, post release.

2. **The Release**: For some projects this may be as simple as uploading a new binary and making it fully available. For others they may incorporate decisions and mechanisms to make each release with the aim of de-risking any undesirable/adverse effects of the new release.

3. **Deployment**: Deployment occurs when end users install and start using the release of the app. Both the app store and the end users affect when this occurs. App developers can try to hasten when users install the latest release through various mechanisms, for instance through implementing and mandating users upgrade their current release.
The graph at the bottom of Figure 2.5 illustrates these three phases together with some of the dynamics, e.g. of rollout and disuse of a release. These phases are part of a longer lifespan of the release that includes an often long-term postdelivery period [41, pp 156-157] where users use the release until it is decommissioned or replaced with a subsequent release.

Figure 2.5 also highlights how these phases of a mobile app release relate to the ‘life span stages’ in [41, p.155]. Mobile app releases in an app store extend the Delivery which may also overlap either or both the development and the postdelivery life span stages. The overlap with the development stage is because the development is not complete until the app store accepts/approves the release (this may include pre-launch checks, automated testing, and so on depending on the app store). The overlap with deployment happens as releases are often released incrementally initially to a small percentage of the userbase - at least some of the users in that percentage will install the new release, until the percentage has been achieved. Meanwhile at least some of those users will use the app which will then mean the release is operational and may need operational support.
2.2 Mobile app software development practices

As noted in the previous chapter, few if any developers write perfect software, and this applies also to developers of mobile apps. In order to improve the quality of their software, mobile app developers need information to support their understanding of the issues affecting their app. Mobile analytics is an important source of information for this purpose.

Figure 2.6 provides a conceptual representation of how analytics information flows are situated in the context of the development practices and artefacts.

At the highest level, the development context is organised into three zones: _development_, _pre-production_, and production. Developers are in the dev zone where they work on tasks and create release candidates of their mobile app(s). The release candidates associated with each of these zones are illustrated by a coloured circle and their role, together with the relevant mobile analytics information flows, can be described as follows:

- Green are releases in the safe zone, within the development zone. These are easy to work with and easy to cease.
- Amber releases are in the pre-release zone, they may be used by people external to the development team e.g. colleagues, senior managers, closed, and/or open groups of users. The releases require some additional management to control the release rollout, some additional support, and they’re harder to cease i.e. to stop.
them from being used. App stores and/or third-party services may help make the releases available to these users.

- Red releases are in the production zone, they can be used by anyone who downloads them from the app store. Once releases reach production they can be extremely hard to cease entirely. In practice users are found who use years-old releases even after any mechanisms have been enabled to prevent those releases from functioning.

Within each release candidate developers can choose to incorporate one or more in-app mobile analytics libraries as part of a mobile analytics SDK. In the diagram the in-app mobile analytics libraries are represented by a small aquamarine spot. As the release candidate moves from development to pre-production usage is likely to increase somewhat, and as the release candidate is promoted into production the usage increases significantly, often by orders of magnitude for popular apps. The purple arcs under the circles represent the feedback being generated by the usage of that release of the app.

There are various forms of arrowed lines in the diagram:

- Dotted lines are for feedback received before the app is in production, so from when apps are in the dev or pre-production zones.
- Aquamarine indicates feedback from in-app analytics.
- Purple is for platform-level analytics.
- Blue is for user-provided feedback. This may be based on using the app or originate from other sources.
- Black indicates the source of information is from testing, regardless of how formal or explicit the testing is. It includes observed behaviours noted by people who are connected with the app developers, e.g. beta testers, colleagues, friends. While these people might also be end-users of the app that is not their primary role when providing this feedback. Automated tests are also a source of this type of information.

Green arcs above the circles represent what can be observed, and purple arcs below the circles represent what mobile analytics can detect and report on. Neither has complete access to all the characteristics or behaviours of the app.

The app developer is shown in green, they are the intended recipient of all the many and various forms of information about the app. They have six ways of dealing with the information they receive. Three of these (1 to 3 in the list below) result in the information not being actioned, the other three (4 to 6 in the list below) result in actions that modify the app. Most of the feedback in terms of volume is likely to emanate from mobile analytics.

1. **Noise, dross, overflow** - the developer does not deal with them at all.
2. **Unactionable** - the developer decides the information cannot be acted upon from a practical perspective.
3. **Purgatory** - the developer placed them in indeterminate hold, for example in case sufficient additional information becomes available to take practical action.
4. **Fix** - where a flaw, including reliability issues, can be fixed by the developer they perform their development process to fix the flaw in a new dev release of the app.

5. **Modify in-app analytics** - for example to collect additional related information in a future release of the app.

6. **Release** - devs can choose to release the latest instance of the app.

### 2.3 Mobile app development lifecycle

To provide some context for this section, Figure 2.7 illustrates a modern continuous software lifecycle including feedback. We can observe several distinct stages in the development and deployment of software and the feedback each stage can provide. In contrast, Figure 2.8 illustrates a similar software lifecycle for Android apps released through Google Play together with the various forms of feedback.

Key differences between typical CI/CD lifecycles and the one for Google Play is the pre-launch testing and the app store providing both user feedback and a service called Android Vitals. The pre-launch reports are generated automatically by Google where the app store runs automated monkey testing on a farm of Android devices and various static analysis checks of releases deployed to any of the test channels. They are described in Test Channels.

There are additional sources of *analogue feedback* from people, including from alpha and beta testers and end users; and *digital feedback* from Google tools and from usage data collected from the field.

---

18. Reproduced from "An empirical study of architecting for continuous delivery and deployment" [42]

19. Here we have excluded feedback from the app store, nonetheless it exists for many app stores.
2.4 Mobile app usage lifecycle

Mobile apps have a usage lifecycle, which starts when an app is chosen to be installed and ends with either abandonment or active removal of the app from a device. Figure 2.9 illustrates the possible stages of a mobile app’s life on a user’s device. Google Play Console collects data consistent with this lifecycle, analyses it and provides aggregate reports based on their analysis.

For clarity and completeness there is another lifecycle when the app is running, described in the Android documentation as the “Processes and Application Lifecycle” [44]. These are more detailed and are not included in the reports Google provides developers. However, it should be noted that the processes and application lifecycle may affect how in-app analytics libraries behave, including when they transmit their data to their respective central servers.

In 2017, Google launched a service called Android Vitals as a new, intrinsic part of Google Play Console [45] that expands usage-based statistics to include the performance of the app while it is being used. As part of Android Vitals Google popularised a measure called Stability to assess the quality of Android apps. Their measure includes both crashes and when an application freezes or is unresponsive for at least 5 seconds from a user’s perspective, a term Google call ANR.

![Figure 2.8: Google Play App Development and Feedback](image)

![Figure 2.9: Mobile App Usage Lifecycle](image)

---

20: Based on a figure in [43]: ‘Falling asleep with Angry Birds, Facebook and Kindle: a large scale study on mobile application usage’

[44]: Android Developers (2020), Processes and Application Lifecycle

[45]: Buch (2017), I/O 2017: Everything new in the Google Play Console
Crashes are often considered a concrete measure of poor performance of software and there has been extensive research in crashes (see Chapter 3 in Section 3.4.6) for Android applications, in particular. It is assumed that the focus on crashes as an oracle for testing software, is because crashes are unambiguous (even if the causes are not) and it is easy to determine whether software has, or has not, crashed.

### 2.5 DevOps for mobile apps

This section extends the DevOps concept of an infinite loop for software generally before becoming more specialised on DevOps for mobile apps. The focus here is on data from various stages of a conceptual infinite combined development and operations process to indicate where mobile analytics applies in terms of providing data to the development team. This data includes: log data, static analysis results, test results, release and usage data, and mobile analytics data and reports.

In October 2020, one of the students taking part in the PhD symposium at the ICST2020 conference presented a variation of the DevOps loop that is relevant to this research. Their focus was on crash reproduction and a reproduction of their illustration is presented with some additional annotations in Figure 2.10. The annotations to this figure show, in red, the extent software can be observed, and in green of when the results of those observations can be applied to the code. Note: the observations can be applied throughout every phase, for instance during deployment aberrant behaviour observed during the deployment may lead to the deployment being paused.

Data is available using various sources about software from the outset of coding until the software dies; however for the purposes of this thesis the focus is on when the software is actively used and maintained, as illustrated in Figure 2.10.

During coding code analysis tools such as static analysis identifies patterns of potential concern in the source code and similar artifacts such as GUI layouts, strings in resource files, etc. Tests can also be created and performed from the outset of the coding to provide runtime feedback (i.e. data) about the software under test, and similarly developers can add logging statements and also use logging built into the operating system that both provide data that can be mined to learn about the software’s behaviours.

The build process may include configuration details, for instance to create a range of custom applications, to include instrumentation, to compress and obfuscate the application binary, and to create debug and release editions of the software. Data about the build process and about what has been built may also be observed and analysed and the products tested and analysed; for example an application binary can be scanned for information leakage in an obfuscated build, and builds can be tested to provide more data about how the software performs.

Note: the release and deployment phases will be covered in more detail in the next section; here the focus is on the data available as part of these phases.
As software is released the software transitions from being within the view of the development team to being used by others where the development team no longer has direct access to the devices that have the app installed or being used. They are remote from the use of the app. This means that local access to devices (to see what’s happening and to try things out) and to the device’s logs are unlikely to be practical, other sources of information now need to take precedence.

The release of a mobile app using an app store is subject to the processes and controls applied by the provider of the app store. The app store may offer both free and paid-for optional services to the developer, for example Google provides optional, free pre-launch reports that contain the results of automated testing and static analysis of application binaries. The app store may provide reports to the development team particularly if they decide to delay or block a release or suspend an app from being downloaded by end-users.

Usage is the ultimate active phase for a mobile app installed on a user’s device. Simplifying slightly, as some apps run automatically in the background, most apps are started and used by end-users. Aspects of the usage can be recorded by various software utilities, in particular by the platform which records when an app starts and when it terminates. The platform can also record when an app is installed, when it is updated, and when it is uninstalled. The app store may provide developers with reports and statistics on the app’s install base and usage. The app store may also provide developers with information about the performance of
the app including any failures of the app while it was running.

Crashes are logged locally by the platform, some platforms may also record other failures and performance related data as well as resource utilisation and various capacities such as battery level locally. The platform may have permission from end-users to forward a copy of the information logged locally on the device and to use it for various purposes.

### 2.6 Software releases for mobile apps

For mobile apps, the release management may include deployment of the app to end-user’s devices either as a fresh install for new users or as an update for existing users that app on a given device. Releases may be acute or chronic in nature. Acute releases are actioned immediately and deployed as soon as practical. Chronic releases may involve alpha (closed-group membership) and beta (open-group membership) testing followed by rollout in stages, for instance starting at 10% of the user-base, then increasing to 25%, and so on until the new release is available to 100% of the user-base. Note: there is no guarantee that the new release will be deployed to the entire user-base, and in my professional experience some users will keep using much older releases for as long as several years after newer releases were made available to them.

The actual deployment and market penetration of a new release depends on several factors which may be outside the developer’s direct control. This particularly applies for mobile apps made available through an app store where the app store and end-users can block new releases being applied. In my professional experience across a range of Android apps, for a 100% rollout it takes about a week for the new release to be installed on 50% of the user-base’s devices, however the range varies from 3 days to several weeks to reach 50%.

Figure 2.11 illustrates a typical pattern of the majority of the userbase migrating from one release of the Pocket Code app to another and yet others remain with older releases during this period of 180 days. Note: Google defines active users as: “The number of users who have your app installed on at least 1 device that has been turned on in the last 30 days” so it’s not necessarily those who use the app in this period. The residues of users who remain on older releases of the app constrains the rate of change of the overall statistics as measured by Google Play Console and Android Vitals. This means that old buggy releases may continue to adversely affect the overall stability metrics for the app for long periods, even for months and years. Conversely, the project team may have addressed some of the causes of poor stability yet don’t have a viable way of proposing users upgrade to the newer releases, and sometimes newer releases may have undesirable features or characteristics from an end user’s perspective.

**Installation base:** the installation base for a mobile app ebbs and flows based on new installs and uninstalls. In current mainstream app stores users do not get a choice of which release of an app they will receive when they receive it, that choice is made by the app store using one or more
2.6 Software releases for mobile apps

Figure 2.11: PocketPaint Active Users 180 days by app version

releases provided by the developer. The app store may allow developers to choose various criteria for current releases, such as a percentage rollout, and then apply that. However, the final install base following a release can still differ because some installations are not sourced from the app store. A key consideration is that upgrades apply to the install base so they occur within a population and do not change the number of installs. That said they may lead to decreases in the installation base for various reasons. Some users may uninstall an app prompted by learning of a new update of the app - for instance if it requires permissions the user considers intrusive or unsafe; others may uninstall the app after the update, for instance if they believe it is too buggy to merit using. Note: they are not able to revert to an earlier release within the current major app store’s practices / constraints.

To sum up so far, new installs receive the most current release gated by the rollout algorithms implemented by and in the app store. Upgrades also receive the most current release gated by the same rollout algorithms.

Cohorts: represent a group of users who share something in common and measures at intervals through time. Examples include new users who install the app on a particular day or in a particular week. They are
used in analysis, for instance to measure what proportion of new users still have the app installed a week later. Cohorts can also be used to group users with a distinct release of an app, and so on. Useful introductions to cohort analysis using Google Analytics are available online [47, 48], and cohorts are used in various analytics tools including Google Analytics and Firebase Analytics.

One reason why users uninstall an app is because of poor quality behaviours by the app. In a survey by Dimensional Research, 3011 participants completed an electronic survey to identify key factors that led to end user satisfaction with mobile apps. The research also ‘...sought to determine what users did when they were unsatisfied with a mobile app.’ [49, p. 5] 26 53% of respondents have uninstalled or removed mobile apps that regularly crashed, stopped responding or had errors and 37% stopped using the app (Ibid. p. 16).

To sum up again, the installation base relies on a combination of existing and new users. Updates can lead to some existing users uninstalling the app - a topic to consider when planning releases of the app. Cohorts can be used to measure the effects of new releases of an app and several analytics tools already use them for tracking the retention of new users. Poor quality of an app may lead to more users abandoning and even uninstalling the app. As a hypothesis: if an update of an app improves the quality of an app for a user the user may be pleased with the update and continue using the app, conversely if an update does not improve the quality of the app for that user some users will merely stop using the app others will actively deinstall it. Releases can be a point of inflection in terms of the userbase.

FORCED UPDATES? Some developers, and some platforms, may incorporate mechanisms to encourage or even try to force users to update their apps 27. However, doing so may upset and alienate some users, and as was observed in case study, C1, there were still users who continued to use old releases with high failure rates.

The Kiwix Android app has over 70 releases being reported as active in a 7 day period, some several years old. Figure 2.12 shows the overall active user base and the top ten of the 70+ releases according to Google Play Console 28.

As mentioned above, there are various practical constraints to the frequency of releasing mobile apps using an app store. Chiefly there are two constraints: 1) the relatively slow rollout of new releases to the user-base which can take a week or more to achieve 50% and also 2) the app store’s review process which has been a hotly debated topic particularly by developers who may end up waiting days or even weeks for a release to be approved. Apple states “Review times may vary by app. On average, 50% of apps are reviewed in 24 hours and over 90% are reviewed in 48 hours.” and Google informs developers in Google Play Console “We’re experiencing longer than usual review times. Due to adjusted work schedules at this time, you may experience longer than usual review times for your app.” 29. Sophisticated development teams may find ways to alleviate these constraints, for instance by shipping code updates that are applied by a current release rather than by creating and releasing a new binary of the entire app.
2.7 Chapter summary

We now have a grounding in the domain of mobile apps, logging and mobile analytics which is underpinned by research in both academic research and grey literature. The conceptual models introduced in this chapter provide an understanding of the relationships between key entities in the mobile app ecosystem, including app stores, developers and end users. Additionally, we have discussed some of the techniques used by development teams to measure the performance of their apps and gain an understanding where software quality issues may be affecting this performance. In the next chapter, we discuss the state of the art with respect to the key areas of investigation relevant to this research: software quality, software analytics, and the mobile app ecosystem.
The research reported in this dissertation sits at the intersection of prior art in several related fields, namely: software quality, software analytics, and the mobile app ecosystem. Figure 3.1 provides a visual overview of the major influences that provided the context for this research. Previously, 2. A grounding in the domain, provided an overview of development practices and introduced various conceptual models followed by various practical aspects pertaining to mobile app development practices. This chapter aims to analyse the state of the art in these fields and identify the pertinent gaps which led to this research being performed, i.e., which motivated me to act.

Research in how mobile apps are created and tested, the relevance of app stores, service and utility providers, the user bases for mobile apps within the overall population of users of an app store ecosystem are all relevant. And meanwhile understanding why it’s hard to create reliable software is also vital as part of acknowledging some of the grim realities development teams need to face if they are to succeed in their other goals and objectives for their mobile apps. An understanding of research into how to measure software qualities, and stability in particular, is key to establishing ways mobile analytics measures these qualities.

At times this chapter will draw from broader sources, for instance in software development, testing, and analytics, as these provide context for the particulars of the mobile app ecosystem. Conversely, the mobile ecosystem is influencing the desktop app ecosystems in different ways. Examples include: app stores, public ratings and reviews, platform (device) level, crash reporting, and usage analytics.

Contents of this chapter: 1 Section 3.1. Literature review method, provides an overview of how the material for how this literature review was sourced and compiled.

Subsequent sections are organised to provide the context of the research. This research builds on several strands of existing research. The first comes from software development, in Section 3.2. Software Development, and includes software development in general in terms of: development practices, software quality, including measuring software reliability, and then the use of software analytics. The second strand, in Section 3.3. App Stores and their Effects on Software Development and Engineering, is the app store ecosystem and its effects on software development and engineering.

These two strands provide the context for the section on developing mobile apps, Section 3.4. Developing Mobile Apps, as they both affect...
development practices when developing, testing, releasing, and maintaining mobile apps. This section includes prior research into crashes and freezes of mobile apps, as these are both indicators of poor quality-in-use of the mobile app.

Sources of information regarding software quality for developers of mobile apps are pertinent to app developers who wish to improve the software quality of their apps. Therefore, Section 3.5. Sources of information on software quality for developers of mobile apps investigates prior research in this area.

### 3.1 Literature review method

Where others have done similar work they’ve done so in other ways, e.g., Mining Software Repositories (MSR) and/or Systematic Literature Reviews (SLRs) rather than focusing on the development teams. While their work is tremendously interesting, it does not take the perspective of app developers. Further there is limited prior research that focuses on the use of mobile analytics or effects of analytics on the quality of the mobile apps produced and their related artefacts.
3.1 Literature review method

Figure 3.2: Overview of the literature review process and outcomes

Figure 3.2 illustrates my approach to researching prior work in the use of mobile analytics by app developers. The initial trawl for prior work was based on searching for scholarly articles and accounts of software development practices that included different combinations covering the concepts of: “software quality”, “reliability”, “stability”, “mobile applications”, “software development”, “software release practices”, “software analytics”, “crashes”, “crash analytics”, “ANR”, and “mobile analytics”. These were also combined with additional search terms including: “app store”, “Android”, and “iOS” to seek publications that were more likely to be germane to the research questions.

These search results led to various additional searches, including keyword, tags, and related items, that were incorporated into the searches. Many of these searches were inductive, to seek information, patterns, nuggets of research interest, and so on. Other searches were stimulated by events and findings in the case studies.

Initial sources included Google Scholar to find the more research-oriented materials, and Google Search particularly for grey materials. Specialist search tools were used where sites provided them; for instance, on stack exchange sites such as StackOverflow, GitHub.com, acm.org, ieee.org, and medium.com, their respective search engines were used frequently. Where practical, copies of material have been preserved privately and backed up using at least one commercial, paid-for, cloud file storage service ² for safekeeping and to preserve evidence.

Multi-selection criteria were used to select material that appears to be of interest, relevant, and plausible. Generally bibliographic entries were obtained, and these where checked for accuracy and completeness. Grey material seldom has a bibliographic entry; these were created by hand, and preserved. Snowball sampling helped increase the breadth of the search results. Termination of searches was based on reaching a point of diminishing returns in terms of new information. Selection was based on relevance to the research and to the case studies, as elaborated below.

The results of the searches were filtered, for example, to remove papers that were not relevant to software development practice and those that

²: i.e. Dropbox.
focused on research tools that were not in use by practitioners. This was based on reading the title, abstract, and conclusions of the paper in the first instance, and exploring the findings in greater depth when appropriate. Similarly research into mobile apps and app stores that pre-dated the introduction of mobile analytics was frequently filtered-out as, de-facto, it did not include these aspects.

In reading the literature various false friends emerged, papers that first appear relevant because of their titles and/or abstracts but turn out to be on very different topics. Knowing about the concept of false friends and having pragmatic strategies to deal with them is important to avoid misunderstandings or misleading application of their work, based on [51, p. 1833]. Through a process of sense-making, cross-checking, and corroborations, various thematic groupings emerged, together with potential relationships between the thematic groups. Based on this analysis, three clearly distinct and vital aspects emerged in the related work: the development practices used by mobile app developers, the artefacts they create and maintain, and the mobile analytics tools they use. These were refined further into six perspectives that highlighted the need to consider 1) current practices (i.e., what is) and, 2) opportunities for improvement (i.e., what might be), for each aspect of using mobile analytics.

### 3.2 Software Development

Mobile apps are developed using similar practices to other modern software projects, however there are key distinctions/differences including: the build, packaging, and release processes (which are relatively similar to those for software apps generally). The design, implementation, and function of mobile analytics also differ for mobile apps compared to other modern software projects. There are also nuances in the effects of software quality as measured by the app store which are important for us to be aware of; these are covered in Section 3.3. App Stores and their Effects on Software Development and Engineering.

This section, and associated subsections, provide context for the more specific domain of mobile apps, covered in Section 3.4. Developing Mobile Apps. This section covers the following topics: Section 3.2.1. Software development practices, Section 3.2.2. Software quality, focusing on reliability including measuring aspects, and Section 3.2.4. Software analytics.

#### 3.2.1 Software development practices

Jez Humble is a well-respected leader in modern software development practices who popularised the concepts of Continuous Delivery and DevOps. In [52], he argued that the benefits of using continuous delivery to reduce risks and transaction costs, to create fast feedback loops, and work in small batches. He also believed it could be applied to any software and any domain. He concluded: “This, in turn, increases the quality of products, allows developers to react more rapidly to incidents and changing...”

[51]: Chamizo Domínguez et al. (2002), ‘False friends: their origin and semantics in some selected languages’

[52]: Humble (2018), ‘Continuous Delivery Sounds Great, but Will It Work Here?’

3: As an observation there appears to be little research into desktop apps, such as those available on MacOS. However, this is outside the scope of this research.
requirements and, in turn, build more stable and higher-quality products and services at lower costs."  

To be successful in applying continuous delivery, Humble identifies the importance of continuous, daily improvement and constant discipline to seek higher performance.

Can we assume continuous delivery is suitable for mobile apps and can be applied by developers of those mobile apps? Not necessarily: in an app store, deliveries are packaged into a software release that needs to be approved by the app store; the release needs to be accepted by end users as a replacement for a previous release of the app; and all this takes time. Existing research into releases in app stores, in Section 3.3.1. Managing releases in app stores, provides context for where and how mobile analytics might make a useful contribution.

### 3.2.2 Software quality, focusing on reliability including measuring aspects

A relatively well-established definition of software reliability is: “the probability for failure-free operation of a program for a specified time under a specified set of operating conditions.”[53, p. 25]. The authors noted in p.33 that software reliability cannot be measured directly, instead it needs to be measured via measurable attributes. These can include frequency of use, Probability of Failure On Demand (PFOD), and/or elapsed time, Mean Time Between Failure (MTBF). The authors assumed reliability would be measured through software testing, which is a common approach to do so.

As an example, early work compared two approaches to software testing: debug testing and operational testing [54]. Their research considered the two approaches including their efficacy at improving reliability of the software being tested. This work challenged the focus on ‘faults’, which they stated was nebulous and not necessarily the best term to use to describe what happened when failures were detected or when changes were made to ‘fix’ the code that led to the failures. They introduced the notion of failure regions, which could be “eliminated by a program change”. This research and associated concepts align well with the use of mobile analytics to identify failures, and failure regions that are detected by mobile analytics and potentially addressed by changes to the program, i.e., the mobile app. This paper is revisited in Section 3.4.6. Mobile app crashes.

The authors discussed the potential perils of relying on the judgement of people involved in debug testing (testing to maximise the faults found in a given period). They referenced [55] where testers who focused on ‘testing’ over-estimated their abilities to find all the faults in a program. To counter the peril, they proposed comparing the effectiveness of testing by using operational profiles [54, p. 77]. Although their research pre-dates mobile apps, smartphones, and mobile analytics, this perspective is relevant to this research because the comparison in terms of the effectiveness of testing could be operational results from mobile analytics reports.

Another germane concern raised by Frankl et al. is the potential for debug testers to confuse “between detecting failures and achieving reliability.” [54,
[54]: Frankl et al. (1997), 'Choosing a testing method to deliver reliability'

[56]: Cândido et al. (2021), 'Log-based software monitoring: a systematic mapping study'

[57]: Murphy et al. (1995), 'Measuring system and software reliability using an automated data collection process'

[58]: Murphy (2004), 'Automating Software Failure Reporting: We Can Only Fix These Bugs We Know About.'

[59]: Dey et al. (2020), 'Deriving a usage-independent software quality metric'

As mentioned earlier, testing is considered a common approach to measuring reliability; in contrast, in [56] "The data provided by application logs and runtime environment are essential to ... improve system reliability". This approach is similar to using mobile analytics, which pre-dates their approach. In terms of gaps, they did not consider the reliability of mobile apps or the use of mobile analytics. Note: Work by Murphy and Gent[57] on VAX mainframes studied the causes of system crashes at customer sites. This was extended by Murphy [58] for a desktop operating system (Windows XP). Their work illustrated challenges pertaining to collecting failure data from distributed, remote computers over intermittent online connections – these are similar to data transfer challenges faced by mobile analytics SDKs.

Google Analytics has been used to record crashes in the Android and iOS releases of several mobile applications developed by Avaya[59]. They found a clear correlation between the number of users and the number of crashes (p.1636). While their work did not discuss whether the developers used the information provided by the analytics nor whether the results were compared with platform analytics, it corroborated that crashes can be used to measure software quality.²

One of the challenges has been to find ways to measure beyond an app, to behaviours that are externally observable but not easy to measure within an app, for instance how long the app takes to start from being launched. Unresponsiveness of software is a particular concern for mobile apps where apps stop responding for longer than deemed acceptable. Google Android coined the term ANR for when the application stops responding. One of the challenges has been how to detect and measure these behaviours, particularly remotely on end-user devices.

3.2.3 Stability and Reliability

Reliability is a well-established term in software engineering, as the previous subsection attests. Stability has also been used to describe the quality of software; however it means different things to different authors. For example in [60] stability is not defined; rather the term infers software models that have the characteristic of being stable, i.e., unchanged. However, no connection is made with software quality in use. Similarly, in [61] stability is both used in the context of APIs in the Android platform and used to qualify the rate of change of an API across releases of Android.

As examples of the the lack of consistency between these terms related to software quality:

1. in [62] reliability contributes to dependability, but stability is not mentioned;

2. in [63] both reliability and stability are considered contributors to dependability; whereas ISO 25000 series do not mention stability or dependability, and reliability is a first-class citizen [64].
In industry, the term stability is used to describe and measure software quality in use. For example it was a software quality identified initially by HP as part of its FunDex score [65]. The same term was subsequently used by Google to identify and measure two related indications of software failures: crashes and freezes [66] in Android apps. Huawei, who established an app store and was a major manufacturer of Android devices, also uses the terms reliability and stability. As this research focuses on Android apps in Google’s Play Store, Google’s use of the term stability is the primary source used in this dissertation.

As this research is in the context of the practices of mobile apps and mobile app developers, the industrial use of the term stability is most appropriate. In this dissertation, stability will be used when the focus is more practical in nature, and reliability will be used when the focus is more research or software engineering oriented.

3.2.4 Software analytics

Software analytics includes the use of analytics to measure and potentially improve processes, products, and software tools. Some of the early published research came from Microsoft Research. For example, Buse and Zimmerman wrote a short paper in 2010 which helped establish the field of: “Analytics for Software Development” [29]. The first figure in that paper is reproduced here as 3.3. As an observation, they derived that figure from a more general business-focused diagram in Davenport and Harris’s book “Analytics at work: Smarter decisions, better results” [67, p. 7], so this figure appears to be broadly applicable once it has been customised.

Buse and Zimmerman “distinguish between questions of information which can be directly measured, from questions of insight which arise from a careful analytic analysis and provide managers with a basis for action.” [29, p. 78]. This is an important distinction. In this research, the insights are also pertinent to app developers, rather than being limited to managers.
Information needs

Development teams comprise many roles, including development managers and software developers. As Microsoft discovered, these two roles have different information needs in terms of software development analytics [30]. They asked 110 Microsofties, a mix of development leads and managers, about their use of software development analytics; none were identified as working with mobile apps or Microsoft's then current mobile operating system or platform (p. 989).

Both the managers and the development leads agreed that “it [is] more important to understand the past than try to predict the future; echoing George Santayana, ‘those who cannot remember the past are condemned to repeat it.’ ” (p. 990). From Figure 4 in their work (p. 991), Failure Information (“reports of crashes or other problems”) is one of the top 3 indicators that the development leads would use, and top for managers when making decisions relevant to their engineering process. Telemetry would be used by approximately 80% of development leads and 90% of managers in the survey. The respondents used crash reports in their work, and presumably these were considered to be part of software analytics, given they were mentioned in the research. The survey did not appear to discuss software quality, reliability, or the use of software analytics (or telemetry) to improve the quality of their processes, artefacts, or analytics tools. They did consider data quality, which was important to the respondents. Participants confirmed that analytics could help them “monitor a project; know what’s really working; improve efficiency; manage risk; anticipate changes; evaluate past decisions.” (p. 989).

“The landscape of artifacts and indicators in software engineering is well known, but the concrete decisions they might support are not. We conjecture that understanding how managers and developers can make use of information is critical to understanding what information should be delivered to them.” (p. 992). My research somewhat inverts this by looking at what’s being delivered in terms of commercial, pre-existing mobile analytics and considers how that information has been used to improve the reliability of the mobile apps that were the source of the raw information.

Overall, their focus seemed to be more on the code than the product of the code (the apps, services, and qualities of the services those apps provide to end users). Unasked and unanswered in their work were: ‘What information do mobile app developers need in terms of improving their practices, the reliability of their apps, and the analytics tools and services they use?’ and ‘How can the developers make good decisions in this area?’

A much more recent paper, published on arxiv.org, performed a SLR [68]. All the accepted 42 studies were aimed at developers (p. 10). Many of the papers included App Stores (p. 10), and the majority of the selected papers were from the Empirical Software Engineering Journal 9, yet none of the papers were identified with mobile analytics or crash reporting, despite these two highly relevant criteria.

[30]: Buse et al. (2012), ‘Information needs for software development analytics’
8: Someone who works for Microsoft.
[68]: Caldeira et al. (2020), Software Development Analytics in Practice: A Systematic Literature Review
9: www.springer.com/journal/10664
Software analytics tools

Software analytics generally involves software that is used as a tool to analyse whatever the data is. The tool may be limited to reporting based on existing data, or it may include data gathering and collection. The tools may range from a spreadsheet editor, through heavy-duty yet data-agnostic tools, to custom and highly-tailored reporting systems. In this research, the focus is on the custom and highly-tailored reporting systems, generally packaged with the respective data collection Software Development Kit (SDK), services, etc. as these are intended and designed specifically for mobile analytics.

In terms of research into software analytics and tools used to provide analytics about the use of software, Microsoft’s work with their Windows Error Reporting (WER) is of particular interest as their research is practical, and is grounded in real-world projects.

Learning from Windows Error Reporting: The data collection and analysis of Android Vitals shares various similarities with Microsoft’s Windows Error Reporting (WER). WER is described from a research perspective in an article from 2011 [69] and a longer conference paper from 2009 [70]. Similarities include both mechanisms being designed to work effectively at a scale of at least a billion end-user machines and devices; data capture including capturing crash data; and the use of error statistics as a tool in debugging.

Differences include the platform (Android), which lacks automatic diagnosis (which WER provided). And, most relevantly, Google provides several million third-party developers with access to data for the apps for which they are responsible and provides them with various comparative analytics of the technical performance of their app compared to those apps of their peers. (Microsoft provided 700 third-party developers, for instance of device drivers, with WER information, and the paper provides a concrete example of how one vendor addressed the top 20 reported issues for their code, and how the fixes percolated out to the end users and halved the percentage of all kernel crashes attributed to that vendor (from 7.6% to 3.8%).) Statistics-based debugging, described in these papers, was used in Microsoft’s WER and may also apply when developers use mobile analytics.

Beyond Microsoft: Microsoft is not unique in publishing research about the use of software analytics. A company, Softeam, used a software analytics platform to collect feedback from its customers and systems with the aim of improving the software quality of its systems [71]. It uses github.com/q-rapids. There are additional publications of interest listed at www.q-rapids.eu/publications, however they do not contribute further to this research.

Interpreting analytics data

As Microsoft noted, interpreting the data in order to provide a basis for action is a necessary facet of what development teams need to do. This
section considers two aspects: connecting, or combining, failures with software analytics, and caveats with software analytics.

**Connecting failures with software analytics:** In a short paper [72], Kidwell et al. propose combining fault classification and software analytics for five types of decisions. These are: targeting testing, release planning, judging stability, targeting training, and targeting inspection of software. Their paper provided initial indicative evidence of their proposals through evaluation of changes to source code for the Eclipse software. It also discussed the measurement of refactoring to provide more accurate and relevant measurements of the efficacy of the refactoring. This work is relevant in terms of aspects such as judging stability and targeting inspection of the software. Their research did not consider approaches to improve mobile apps through mobile analytics.

Failure data mined from software analytics tools such as crash reporting tools might help to bring their concepts and ideas to life.

**Caveats with software analytics:** Using software analytics leads to several caveats to consider the ‘so what’ aspects and whether the research is being done well. For the research reported in this dissertation to be applicable, it needs to pass two tests. The first test is to answer: “Software Analytics, so what?” [73]. The second is to avoid doing or publishing poor quality work [74]!

### 3.2.5 Summary

This section has identified prior art in software development practices including the use of software analytics for software projects; while a few examples are for mobile apps, the majority of examples are not – they are for other platforms or more general in nature. They may help us understand ways in which app developers can improve the reliability of their apps through their practices. The gaps in knowledge in this section’s material include mobile app development practices, a topic investigated shortly in Section 3.4. Developing Mobile Apps, and sources of information on software quality for mobile app developers, which will be covered in Section 3.5. Sources of information on software quality for developers of mobile apps. Before doing so, the next section will consider app stores and their effects on software development and engineering practices.

### 3.3 App Stores and their Effects on Software Development and Engineering

App stores are part of an ecosystem that provides various services; the app stores also enforce rules that constrain various choices. They also allow various freedoms for the participants within the ecosystem. The ecosystem needs to sustain competition [35, p. 94] and revenues. The two largest app stores, Google Play and Apple’s App Store, each has its own development platform including an Integrated Development Environment (IDE) and additional software tools, various SDKs, developer
programs, release processes, pricing and revenue rules, and so on. Pivotal effects include: the app approval process (which gates any release of the app to the general user population), rules and restrictions on what the binary contains, the signing process, how the app is packaged/bundled. Another pivotal effect is how app quality is measured and assessed by the app store, at least, as app developers have a vested interest in having high quality apps as determined by the app store.

App Stores behave as intermediaries between developers and the users of their software. They make various aspects more transparent including pricing, information about the apps, releases, and ratings and reviews. There are hundreds of thousands of developers of Android apps according to various sources (320,000 in 2017 [75]).

In an App Store, first the developer then the app store is involved in making a release available to some or all of the user population. Various competing factors influence when it is optimal to make a release. If the release frequency is too low, an app may be considered stale or neglected. On the other hand, if the release frequency is too high, users may balk at the seemingly endless updates and communications costs. This topic is covered in Section 3.3.1. Managing releases in app stores.

Their research focuses primarily on the Android ecosystem and the Google Play store – the combination is the preeminent platform in terms of userbase, reach, and platform analytics provided to app developers. However, recognising that these are other ecosystems, their research also includes investigations of several additional platforms and app stores (e.g., the Window Phone platform with Microsoft’s app store, and Huawei’s app store for Android) where they contribute to their research.

App stores and their ecosystem have affected the lives of billions of end users and millions of software developers. They have become the primary route-to-market for many app developers and their organisations (exceptions include companies who developed strong businesses elsewhere such as Amazon and Netflix). From a research perspective, in 2010, early papers were published on various effects of app stores on academic research, e.g., how app stores addressed some of the previous constraints such as reaching more users and facilitating the distribution of the apps and feedback from those users.

Cramer et al. discussed aspects of research in the large and in particular the importance of “playing by the rules” [76]. Their research identified the importance of what happens when developers were deemed not to play by the rules (covered in Section 3.3.2. Power dynamics). It should be noted that Cramer et al. shaped their research to play by the rules of the app store.

Miluzzo et al. introduced other relevant research aspects, i.e., ongoing concerns such as how to assess correctness when there is no “ground truth” – a challenge when evaluating mobile analytics for shipping apps; and a software development model of “deploy-use-refine” [77], where app development refines the app based on data gleaned from usage of the app. Their paper even explained how a silly mistake caused their app to crash, whereupon the app store delayed the new release of the app by several weeks. Even in 2010, crashes adversely affected the app store’s perception of an app.


[77] Miluzzo et al. (2010), ‘Research in the app store era: Experiences from the concrete app deployment on the iphone’
In more recent research, Wang, Li, and Gao [78] provided a helpful longitudinal evaluation of the Google Play ecosystem and raised interesting questions and observations about Google Play. However, they did not seem to consider flaws, or the effects of flaws, in the app store’s data collection, algorithms, etc.

In terms of effects of app stores on software engineering practices, there have been several seminal papers. Of these papers, the first [28], combined interviews with a survey to ask developers about their experiences of developing mobile apps, and how those experiences differed from developing other software. Of the ten developers AlSubaihin et al. interviewed, two were for popular apps (800,000 downloads, 2,000 ratings, 1,000 ratings); the rest were for less popular apps.

AlSubaihin et al.’s research identified automatic in-app crash reporting as the most frequent source of bug reports and the second-most frequently addressed in terms of bug fixes (end-user written bugs were addressed slightly more frequently) [11]. Of particular interest was the discovery that, according to the survey results, the quality of the [source] code was the least important factor to build a successful app, and furthermore the number of downloads was the highest measure of success [12]. Note: Google has subsequently (but not necessarily because of their research) placed a lot of focus on encouraging developers to improve the quality of their Android apps in Google Play.

AlSubaihin et al.’s study is one of several that discusses the gaming of app store ratings. Such gaming is unsurprising, given the importance placed on these ratings and particularly in having mobile apps with high ratings in the app store. Given their importance in the mobile app ecosystem, ratings and reviews are one of the topics covered in Section 3.5. Sources of information on software quality for developers of mobile apps.

Reviews can be mined for relevant information (including bug reports), which can then be used alongside user ratings in assessing goodness. Figure 3.5 is an annotated version of their Fig. 2 [14]. The annotations include feedback in the forms of ratings and reviews, and in the form of mobile analytics. Various sources of information can be used by the development team; of these, ratings and reviews are broadly researched, whereas platform-level and app-level analytics have not been researched previously in the context of app stores.

A key challenge identified by Nagappan et al. is restricted access to data held by the app store. To workaround this restriction, they acquired historical information by mining the app store on a regular basis [15]. Perhaps the authors weren’t aware that developers of Android apps have access to various historical data about their apps, including long term access to all the reviews of their apps. Details of how developers can download these and other reports, including the data structures, are available online [80].
An area their work did not discuss is whether failure data could, potentially, be a source of requirements, in the same way that app store reviews could be used to extract requirements. Similarly, they did not consider whether complaints and failure data could be combined to help developers prioritise issues they should consider addressing. Finally, in terms of their paper [79], they included two stakeholders: the developers and the end users. Their research did not include the app store – and the people and organisation that provide the app store – they are also direct stakeholders in the ecosystem. There are additional indirect stakeholders including advertisers, researchers, and probably many others.

So far, this review has considered various aspects of app stores and their effects on software development and engineering. Conspicuous by its absence is research into the provision and use of platform-level: mobile analytics, software release management tools, or pre-launch testing.

There are several additional topics that are germane to app stores and their effects; these are covered in the following subsections:

1. Managing releases in app stores covered in section 3.3.1.
3.3.1 Managing releases in app stores

Researchers have investigated various aspects of release engineering, including Adams et al. [81] who argues the relevance of modern release engineering for both software development teams and researchers. Nayebi et al. compared the perspectives of users and developers on release practices for mobile apps. The factors highlighted by this work in includes the optimisation of releases and establishing a successful release strategy, their research and additional research are discussed next.

Adams and McIntosh set the context for the why researchers need to investigate modern software release practices [81] - these release practices apply to mobile apps made available in app stores. They argue it is vital for “empirical validation of best practices and the impact of the release engineering process on (amongst others) software quality is largely missing and provides major research opportunities.”. The authors argue for research into the effects of the release process on software quality (ibid. p.86) but stop at that point rather than performing any such research. Their research provides a really helpful external overview of the field of modern software practices; it does not extend to the actual empirical research they argue for. This dissertation includes empirical research into software qualities including reliability and also presents some of the release management and tracking provided in Google Play Console and Android Vitals.

In follow up research, Nayebi et al. [82] investigated which versions of opensource apps should have been released to the appstore by comparing commits to the source code repository to releases in Google Play (some releases to the app store contain multiple updates through multiple commits to the repository 16). They identified four factors: user-feedback, team-feedback, social-media, and sales; so three sources of direct feedback from humans and then income. Similar work by [83] aimed to identify a ‘release opportunity’ that maximised the positive feedback from the userbase. They used the release frequency of live apps in Google Play as their primary data source. Additionally they factored in: app ranking, rating trend, and update purpose which all played a role in affecting the update results 17. Their work does not use signals such as the stability of the app. They also claim: “Additionally, app quality can be unstable with fast [release] iteration[s].” 18 In their conclusion they noted that bug fixing releases are “more welcomed” 19.

An alternative perspective is provided by Guilardi et al. [84] who switch from exploring when developers should release their apps to asking how well developers keep up with new releases of Android. This is a relatively recent paper where the researchers discovered that developers are slow to revise and update their Android apps for new releases of the operating system. Some of the apps have flaws exposed when running on new versions of the operating system. As a real-world example the release notes for Android 7.0 (Nougat) explains crashes may occur on configuration changes. It exhorts developers to make their code robust and to test their app can survive the new behaviours. 20. For apps to

---

16: From a practitioner’s perspective this is standard practice, their aim was to help identify when to create a release, albeit in hindsight
17: These terms were used six times in the 10 page paper.
18: [p. 2, Ibid.]
19: [p. 10, Ibid.]
81: Adams et al. (2016), ‘Modern release engineering in a nutshell–why researchers should care’
82: Nayebi et al. (2017), ‘Which version should be released to app store?’
84: Guilardi et al. (2020), ‘Are apps ready for new Android releases?’
20: https://developer.android.com/about/versions/nougat/android-7.0-changes#other
retain their quality they need to be updated, new releases of the operating system are one such reason. Other reasons to create new releases include: new releases of libraries included by the app, new contexts of use, etc.

None of the research into managing releases in an app store considered mobile analytics as a source of information, leaving a gap which the research reported in this dissertation begins to fill.

### 3.3.2 Power dynamics

Grey literature is the primary source of evidence regarding the imbalance in power between app developers and the app store, at least in terms of Google Play and Android apps in that store. Four articles are discussed here as illustrations of the imbalance. These four articles, by four different developers, have been selected from the 20+ related articles published on Medium about how they lost access to their apps and in some cases their account.

The termination of a personal Google Play account for an indirectly associated developer, via one of the employees of the small company, was attributed to the termination of that company’s Google Play account [85]. This story has ongoing updates that list various developers who have had similar experiences. A similar experience was reported where someone’s associated account was believed to be the cause of another person’s account being terminated[86]. As Méa dry noted “google@play-store: sudo rm -rf org.mtransit.android” which happened abruptly within a few hours of Google sending automated email notifications to the developer. The reason given in the automated emails was that the app was in violation of the “Deceptive Behavior policy” [87].

In these three instances the developer accounts were reinstated. In Marcher [88] the account was not reinstated and two of his clients also had their accounts terminated. As Marcher noted: “I develop software not just for fun but also primarily for a living. This action not only deprives me of a substantial part of income, but it also forbids me for life to continue my work which is also my passion”.21

The imbalance of power is a systemic effect that can affect both developers and researchers, in terms of their use of mobile analytics.

### 3.3.3 Summary

App stores and their broader ecosystem, illustrated in figure 2.1, clearly affect software development and release practices of app developers for that ecosystem. The research presented in this section illustrated ways developers can manage their releases in app stores and the imbalance in power dynamics between developers of the apps and the provider of the app store. None covers the sources of Platform analytics and in-app analytics in an app store ecosystem, nor how developers can use these sources of analytics to improve the reliability of their apps.

The next area of research is in developing mobile apps to investigate what pertinent research exists.
### 3.4 Developing Mobile Apps

Research into the development practices and mechanisms for mobile apps establishes various norms, characteristics, and habits of app developers whose work, collectively, is used by billions of people throughout the world. In terms of their research, understanding the development practices and mechanisms is vital. Developers also have a voice, and hundreds have been interviewed to learn what’s important to them, for example in [89]. They also communicate through various artefacts they create and maintain; again these artefacts have been studied to learn about their work, for example in [90].

Mobile apps need to be made and developers make them. There are various working practices, apps are made by visionaries, employees, amateurs, and communities. There are various activities involved including development, testing, release, and deployment.

Pascarella et al. studied 5,000 commit messages (of over 1.8 million identified) were studied that were selected from 8,280 active opensource Android apps where the source code repositories were on github.com. The top three activities were 1) app enhancement, followed by 2) bug fixes, and then 3) project management [90, p. 144]. 35 of the commit messages were attributed to addressing crashes in the app, according to their figure 3, on p. 151. They did not investigate how developers had discovered issues, nor whether mobile analytics were affected in the commits, leaving these aspects unknown and unreported.

Research by Joorabchi et al., published in 2013, identified several key topics of concern for app developers, including a strong need for monitoring and analysis support, for instance to monitor the health of an app. Similarly, a major problem was crashes “which are often intermittent, non-deterministic, and irrecoverable.” [89, p. 21]. Interestingly, their research was one of the first to highlight the need for App Stores to provide testing APIs 22, foreshadowing the functionality of Google Play’s Pre-launch reports.

The rest of this section includes the following areas of research related to developing mobile apps:

- Section 3.4.1 provides a brief context for the rest of the subsections.
- Section 3.4.2 considers developers’ use of third-party software libraries, often to enable Section 3.4.3. Services used by apps.
- Developers also use services, including hosted devices for pre-launch testing of their code Section 3.4.4. Services used by developers.
- Many app developers test their apps, research into testing for mobile apps is in Section 3.4.5. One aspect of testing is to try to find and/or reproduce crashes.
- Crashes can be a bane for app developers, Section 3.4.6 discusses research into crashes and Section 3.4.7 extends the topic into the application of crash analytics to automatically find and fix the causes of the crashes.
- Another bane is when the apps freeze and/or stop responding, Section 3.4.8.
- The final topic in this section is the maintenance of mobile apps, raised in Section 3.4.9.
3.4 Developing Mobile Apps

3.4.1 Utility and Service Providers

In the context of this research, mobile analytics are provided to app developers as utilities and/or services by third-parties; therefore it is useful to learn of pertinent research into the use of these utilities and services. Developers use software and related services from various providers where the providers are the primary source of any updates to the software and/or services that they provide to the developers.

Utilities include software libraries, services used by apps, and services used by the app development team. Each of these is considered in turn next.

3.4.2 Software libraries

A recent systematic literature review by Zahn et al. provides a useful overview of research on third-party libraries used in Android apps [91].

Android apps incorporate software libraries extensively [92]. The extensive use of software libraries, generally provided by third-parties, indicate the developers place significant trust in these third-party developers. Also, their apps contain large volumes of code that is unlikely to have been tested by the app developers, apart from some sanity and/or smoke testing. Flaws and instabilities in these libraries may be latent until the app is being used at scale by end users. How will the developers learn about the effects of emergent flaws and instabilities?

Examples of why developers choose to use external libraries include to generate revenue (ads) [92, p. 407], and to ease the management of HTTP requests [93, p. 73]. Related work found that 98% of android apps used third-party libraries [94, p. 218], and on average 41% of android apps use code from common libraries [92, p. 409]. The data in Fig. 5 of this paper (on p. 409) highlights that 5% of the apps included Google Analytics and 2.6% used Flurry analytics. Note: Industry data corroborate that mobile analytics are now incorporated in the vast majority of Android apps [95].

Developers delay updating third-party libraries in their Android apps on average by 324 ± 1 days [96, p. 363], even though 85.6% of the libraries could be upgraded by at least one version without modifying the app code [97, p. 2187].

3.4.3 Services used by apps

Two of the extremely popular categories of third-party libraries are advertising (ads) 23 and mobile analytics 24. In some ways, app developers appear to be similar to their end-users – neither group wants to pay up-front for what it uses. Typically service providers have at least one form of unpaid service offering that is either time- or volume-based. Many also have at least one paid-for service offering that has fewer restrictions than their unpaid offering.
3.4.4 Services used by developers

Device testing services, also called remote device farms, are good examples of services that some developers (and/or their organisations) pay to use. They have been available commercially since around 2008. A relatively dated paper about the then-current device farm services provided a useful overview [98] but did not discuss pricing; and it pre-dated the now-popular service offerings by Google, Amazon, and others.

Over the years different offerings have peaked and then either been acquired, retired, or allowed to disappear. Early research, written by one of the co-founders of a device testing service, TestDroid, provided insights into the bug-finding abilities when tests were run on a mix of their devices [99]. (TestDroid became Bitbar, which was later acquired by SmartBear Software.) More recent research investigated the use of an in-house device farm based on the Open-STF project [100] and found that device farms helped the developers test and deliver their mobile apps to their clients.

Google, Amazon, and Microsoft offer paid-for device farms, as do various specialist businesses including SmartBear Software. There have been a couple of public-good initiatives including Open Device Labs 25; and Open STF [101] which is based on a set of open-source projects 26 and enables teams and organisations to build their own device farms or use commercial offerings based on these projects 27.

3.4.5 Testing Mobile Apps

A substantial amount of research energy has gone into trying to find better ways to test mobile apps, ranging from autonomous tools by which the researchers endeavour to deliver better coverage than a small free utility called ‘Monkey’ (and sometimes manage to do so), through to ways to generate automated tests from the text in reviews. Virtually none of these endeavours seems to make any difference to the testing developers do; they don’t use the tools developed from these research efforts.

In terms of developing alternatives, one called TEMA used model-based testing [102] which found unusual bugs not found by Monkey, however the time and effort spent using TEMA might have been found more bugs in less time 28. Dynodroid [103] achieved 55% code coverage compared to 53% using Monkey (both beaten by humans who achieved 60% code coverage!). More recent research compared Monkey and human testing of 62 Android apps and found Monkey to be surprisingly effective fast and effective in crashing even popular commercial apps [104].

User feedback has been incorporated into automated testing of Android apps to help provide context and how and when crashes occur, for example [105]. Their work is indicative of what might work in terms of integration. In the context of this research their work demonstrates how several sources of information can be combined with the aim of helping app developers understand the conditions where crashes occur so the crashes can then be fixed [more easily]. A promising yet ultimately frustrating piece of research by Lii et al., [106] claimed to successfully generate automated tests that reproduced crashes reported in reviews of...
Android apps. Unfortunately, while they described the text-processing work in detail the research omitted any details about the automated test case generation, scripts, or execution.

Various papers, including the following two papers, aimed to provide a systematic (not necessarily comprehensive or complete) review of literature on testing of mobile apps. The first of the papers used more general search and selection [107] and the second aimed to narrow the focus to full papers on research in testing of mobile apps [108]. They provide useful summaries of various forms of testing however the use of mobile analytics was not studied in either of these papers.

There has been a lot of research into various artefacts pertaining to mobile apps. Some of the artefacts (for instance ratings and reviews) are mainly generated by end users, and others (such as source code) are mainly generated by software developers. As well as source code, another artefact is a bug report, also known as ‘tickets’ or ‘issues’ (e.g., github.com, uses ‘issues’, and developers create bug reports as issues on github.com). Bug reports are a hybrid artefact, as potentially anyone can create them, including users of the code and the development team.29

Kaasila et al. provided a quick summary of pertinent research in testing mobile apps; they described how their commercial device testing service was able to find various types of bugs through the use of a variety of mobile devices with different capabilities and performance characteristics. The testing was actually performed by development teams [99].

Asking the developers how they test their Android apps helps to set the context for the utility of much of the research into testing mobile apps. For example, in Linares-Vásquez et al. [109, p. 617], a fairly typical response was: “I mostly do manual testing due to the limited size of my apps. I sometimes use a custom replay system (built into the app) to duplicate bugs after I come across them. This method is usually combined with manual testing (printing debug information to the log) to pinpoint the cause.”

In contrast, researchers wanted to determine whether mutation analysis was effective at testing Android apps [110]. It was somewhat effective, however very few app developers are likely to use mutation testing at all. Therefore their research is unlikely to be relevant in terms of helping app developers improve the quality of their apps in practice.

More relevant research investigated whether it was possible to mine Android crash fixes without access to any issue or change-tracking systems [111]. Their research is covered in more detail in Section 3.4.6. Mobile app crashes given the relevance to app crashes, rather than repeating it here.

“A Large-Scale Study of Application Incompatibilities in Android” [112] promised some interesting run-time issues discovered in their research where the Android version would be a likely cause. However the reproduction package lacked the test scripts or means to reproduce their testing or bug detection. Also, the relevance of their research is diminished by their finding that Android had addressed these backward compatibility issues: “Yet newer versions (since API 24) had no run-time compatibility issues with apps created in the studied span.” (Ibid. p. 222). Their work may well have merit for the research community; unfortunately, it does not appear to have much relevance to developers of real-world Android apps today.

29: That said, for the projects studied in their research they mainly used ‘issues’ and these were mainly created by the project’s extended development team (an extended development team may include people involved in project management and/or testing).


[107]: Zein et al. (2016), ‘A systematic mapping study of mobile application testing techniques’


[109]: Linares-Vásquez et al. (2017), ‘How do Developers Test Android Applications?’

[110]: Deng et al. (2017), ‘Is Mutation Analysis Effective at Testing Android Apps?’

[111]: Cai et al. (2019), ‘A Large-Scale Study of Application Incompatibilities in Android’

[112]: Konget al. (2019), ‘Mining Android Crash Fixes in the Absence of Issue- and Change-Tracking Systems’
Linares-Vásquez et al. proposed a testing framework they created and called MonkeyLab. MonkeyLab mines recorded executions to guide the testing of Android mobile apps. Their approach records GUI events (click events). Members of the project team (developers, testers, etc.) perform the actions, but the authors claim that their log collection process could scale to collecting logs from ordinary users. Key limitations include events that aren’t purely dependent on the user’s GUI inputs. There would also be challenges getting users to accept such an approach where the app records every input they made. Also, they generate GUI events that have x,y coordinates – absolute positioning that may have limited portability to other devices, screen rotations, and so on. Their playback also appears to require rooted devices. There are numerous other limitations described in their paper, including their use of students as developers; nonetheless their work shows promise in terms of detecting and generating patterns the students did not find. It would be interesting to compare the results using accomplished software testers with experience and expertise testing similar Android apps [113].

There has been a tremendous and sustained research interest in software testing, for instance testing is one of the most popular topics at the ICSE series of conferences 30 and the focus of entire conferences including AST 31, ICST 32, and so on. Similarly, the application of software testing to mobile apps is a rich topic with sustained interest in the challenges and facets of testing mobile apps.

The facets include automated testing and automated bug reproduction, maximising the ‘bang for the buck’, for instance in selecting which device models would be most valuable to use with finite testing. Understandably, given that many of the authors work in research rather than industry, the vast majority of the research is on software apps to which they have access, software for which they can obtain the source code (particularly open source), software they can write – conducted with the people available to them (other researchers, students, voluntary participants, and people paid to perform specific tasks). Minute amounts of the work are based on mature, popular software with semi- or fully-professional developers and development teams. Some research projects, particularly CRASHSCOPE [114], offer the potential to reproduce some of the crashes reported by Mobile Analytics if the tools are sufficiently available and current to actually use.

Prioritising devices to test on

The nature of mobile devices means each model is distinct and has different characteristics from other models 33. Some failures are specific to a subset of devices; the subset may have some common factors such as the manufacturer, the firmware, the chipsets, screen resolution, and so on. “Thus even if the app is tested on one device, there is no guarantee that it may work on another device.” [79, p. 27]. Sadly, Nagappan et al. did not provide any evidence for this statement. This corroborates the author’s professional experience that developers have found it prudent to test on a range of device models as soon as there is more than one device model in a mobile ecosystem. Testing an app on all the device models is impractical and probably infeasible, so there has been various research into prioritising which devices to use to test an app.
One of the research papers close to the area of the research in this dissertation uses usage data for two popular app categories (games and media) gathered through a popular Android management app in China [115]. Lu et al.’s work used an operational profile to prioritise the device models to select to test both new or existing apps. The management app, called Wandoujia 34, is used by “500 million people to find apps they want” 35. Daily usage of the top 100 apps in the two app categories was collected for various device models. In various ways, the Wandoujia app management app provides similar capabilities to Google Play, including tracking when apps are installed, and in use. The recommendations are coarse-grained. The research measured the accuracy of their predictions for recommended devices with the actual devices that the app ended up being used on once the app had been launched.

Their work demonstrates that usage data for several app categories was useful to guide developers on the most popular actual device models for their app. They acknowledge several limitations in their work, including their use of incomplete measures for usage, such as foreground network activity, which don’t suit apps that either perform network processing in the background or don’t use the network. Other app management services, particularly Google Play, could provide similar guidance to app developers. And indeed, as Google Play collects additional data for the entire apps store, it could cover some of the gaps and limitations identified in this research.

In research by Vilkomir et al., 24 device-specific faults were found when testing 15 Android apps on 30 Android devices 36 during research to determine how many devices would achieve 100% effectiveness in finding the faults. They found 13 devices were sufficient and they achieved approximately 90% effectiveness with permutations of 5 devices. They obtained the best bug finding results was a spread of Android versions [116].

Neither of these studies made any use of analytics available to developers of the apps; nonetheless their respective approaches could be driven from analytics provided by Google Play Console, assuming the researchers are able to obtain access to those analytics. Test prioritisation could be driven using mobile analytics, but there does not appear to be any published research on this approach.

3.4.6 Mobile app crashes

Crashes in mobile apps have garnered a great deal of research, perhaps as crashes are definitive and relatively easy to detect.

Kong et al. conducted some interesting large-scale research into analysis of various releases of production Android application binaries [111]. The researchers exercised (tested) a large range of apps seeking crashes of the app using an oracle of various patterns of log messages read from the Android device’s log file. They queried the log using the standard Android `logcat` utility. They also combined their dynamic approach with using static analysis tools to identify potential flaws that would lead to crashes of an app. They then tested newer releases of the same app. If the newer version did not crash they analysed the binary files (the Android Package [Kit] (APK) files) for both releases to identify differences in the compiled code that might have been responsible for ‘fixing’ the...
Thus limiting the value of their approach for app developers.

They limited their work to Android framework specific crashes, and excluded app-specific crashes. They devised ways to identify changes that appeared to fix the particular crash(es) they triggered in the earlier releases and generated patch files based on these changes. These patches were then applied to the older release of the app and the app then tested with the same test inputs and runtime environment (at least in terms of using a consistent Android Emulator (also known as a virtual device)).

They provide a relatively detailed replication package online. The supporting website includes scripts and log extracts for the crash reproductions; however, it lacks the mechanisms for generating diffs, applying them, or building the patched APK. The lack of these mechanisms makes the efficacy of their approach hard to reproduce.

Their approach is innovative and could help real-world developers of Android apps to identify and apply snippets of code to reduce the likelihood of their app suffering the same crash. Their 17 fix templates could act as guides for Android developers, and they could potentially be implemented into code-quality tools.

However, their approach only applied for framework-specific crashes, and their choice of runtime environment meant they could only install 56% of the APKs. There are many other sources of crashes, and also apps that include native code (several of my case study apps do). Also their testing is limited to automated ‘monkey’ testing, which may further limit the crashes their approach can find in production apps, particularly those that incorporate user accounts, user-specific content, behaviour, online purchases and many other forms of activities.

It also does not appear to test for crashes related to third-party libraries, e.g., OkHttp, which is extremely popular in Android apps; however potentially this approach could be extended to do so.

In summary, the approach proposed by Kong et al. has the potential to mine crash stack traces (which are available to the developers of the particular apps) to help with aspects of reproducing a subset of those crashes which pertain to Android framework-related crashes. Similarly, it appears it could complement the automated testing provided by Google as part of the pre-launch reports available in Google Play Console and other services.

In contrast, Wang et al. evaluated a mixed set of industrial and opensource test automation tools against popular Android apps available in Google Play. They measured code coverage and the count of unique crashes each tool could find in these apps. They considered more practical aspects, such as finding ways to combine tools to increase the code coverage and fault detection. They also measured the effort required to set up each of the tools to test the industrial apps. Their reference tool was Android ‘Monkey’, the ubiquitous tool shipped with the Android SDK and one of the most widely-used of these tools. This research appears practical and well-grounded. However, it does not compare any aspect of crashes in the wild (i.e., those that affect end users) and some of the crashes that were detected (such as those reported when Sapienz was used to test the Wattpad app) might have established sufficiently unusual conditions for those crashes to be unlikely for most end users. Nonetheless, these crashes might still be of interest to the app’s developers.
The authors did not mention whether they reported the crashes to the app developers, which is a pity, as it would have been interesting to learn whether the app developers were willing and able to fix the crashes. If the researchers had reported the crashes to the app developers then their research – if combined with the Wang et al. research in the previous paper [111] – could have been enlightening to measure at a black box level to determine whether the crashes had been fixed successfully once they had been reported. Mobile analytics could be a possible source of measurements.

Software telemetry has been used to investigate crashes in Android apps; that is, the stability of Android apps has been measured using telemetry data collected by a centralised crash report management service [118]. Roughly one million stack traces were analysed from thousands of Android applications. A subset (over 500,000) of these stack traces was associated with risky API calls, and these were analysed to identify the most common failure reasons. The top five reasons were attributed to:

- memory exhaustion,
- race conditions,
- deadlocks,
- missing resources, and
- corrupt resources.

The authors provide a set of recommendations they claim may help address various classes of the crash failures. However, as these recommendations do not appear to have been tested for their efficacy, their recommendations are theoretical rather than empirical.

There’s an odd claim in the paper on page 1818: “In addition, the platform of Chen et al. (2011), which is based on remote resource management, can make applications require less memory and resources. Hence, it can eliminate the well-known “non-responsive” exceptions in Android.”. For this claim to hold true, all the causes of non-responsive exceptions (or did the authors mean ANRs? they don’t define this term or use it elsewhere in the paper) would need to be a) related to remote resources, and b) the difference would need to be directly related to the amount of memory and resources. Google provides five common patterns for diagnosing ANRs [119, diagnosing_anrs]; none of these mention remote resource management (even if they may contribute to ANRs).

Finally, the authors say in the introduction to section 5.1 ‘API Recommendations’, on page 1813: “Finally, we provide the frequencies of the representative signatures to show how many crashes could be avoided based on the following solutions.” However, they did not appear to provide this, unless they are the charts in their Appendix 1 and if their proposals solve all the causes of the crashes in each category. This was promising, thought-provoking, and interesting work which sadly lacked evidence that their proposed recommendations actually work in practice for any of the apps that provided the crash data or for other Android apps (if their work is generalisable).
3.4.7 Using app store crash analytics to automatically find robustness and reliability failures

A breakthrough stream of work and research by Ravindranath et al. [120] was developed for Microsoft’s now defunct mobile app store for Windows Phone devices and included mining 25 million crashes* that occurred in 2012 on end-user devices when using various apps on their Windows Phones [40]. Ravindranath et al. determined that the top 10% of error buckets covered more than 90% of crashes [41] and discovered that a significant proportion of these stemmed from root causes they could generate externally. For example, they could configure a network proxy to return a HyperText Transfer Protocol (HTTP) 404 response to a network request for network calls made by any of the mobile apps [42].

They built a service called VanarSena to run in the cloud on lots of Windows Phone emulators with the purpose of automatically testing Windows Phone apps. The service automatically instrumented these apps, for instance to add a global Exception Handler (the complete set of five injected modules is described in pp. 193-194). The app is exercised using autonomous automated testing tools called monkeys, and the service includes a range of Fault Inducing Modules (FIMs) [43] that provide inputs and conditions including those that have caused existing apps to crash.

Their work built on earlier work of some of the authors which was performed at a smaller scale on Android [121]. They share a common instrumentation framework implemented first for Android then for Windows Phone.

The authors proposed that VanarSena could be provided by an app store to help test new releases before they were released in production. The automatic testing helps establish whether the apps handle non-ideal conditions without crashing. They used a concept of ‘crash buckets’ and a) identified commonalities in those crash buckets, and b) found that a significant subset could be triggered through manipulating inputs, conditions, and responses to the app.

The results of their research included finding that 1108/3000 production Windows Phone apps had failures that were detected by VanarSena. They uncovered 2969 distinct bugs in these apps, including 1227 that were not previously reported [120, p. 202]. Some of the failing apps had been developed by professionals, others by amateurs – this indicates that developers often write code that does not cope adequately with non-ideal circumstances.

Several of the authors of the 2012 paper [121] collaborated with other colleagues at Microsoft Research to extend the work. In Chandra et al. [122], they describe various improvements to their Caiipa service that delivered eleven-fold more crashes and eight-fold more performance problems [122, pp. 37-38]. The core of their paper presents their plans for a sophisticated automated testing system for testing Windows Phone apps, together with their goals and challenges. The most pertinent goal in terms of this research would be actionable reports [122, p. 36] together with their use of context data and historical data.

*Note: The 25 million crashes were a small subset of the total crashes on end-user devices were a subset of the total based on their third footnote “The developer has no control over the probability.” [120], p. 199].
3.4 Developing Mobile Apps

Given the nature of Microsoft’s objectives, they did not consider Mobile Analytics or crash reporting SDKs (which include breadcrumbs and the ability to report caught errors, etc.). Nor do their papers mention how the app developers perceived their various tools and services. It’s unclear whether they actually involved app developers in their work. Also, given the demise of the Windows Phone platform and app store, it appears much of this work has disappeared without a trace.

3.4.8 Mobile app freezes (ANRs)

An opensource project, ANR-WatchDog 44, provides Android developers with a mechanism to detect and report ANRs in their application. It includes support to report the ANRs to various crash reporters 123. (Two of the four listed (Crashlytics and HockeyApp) have been acquired by Google and Microsoft respectively.) Note: In 2012, Google was asked to provide a facility to detect ANRs from the application 45 so developers would be aware of them and be able to address them. The issue was marked obsolete by Google in 2014 without comment. Yet, approximately six years later, in 2020, Google Android launched an API call that apps can use to determine whether the app was previously quit by the operating system (see chapter 8, Section 8.1.1, starting on page 183 for more detail).

3.4.9 Maintenance of mobile apps

“The area of software maintenance is one of the most researched areas in Software Engineering. However, due to the fact that mobile apps is a young subarea within SE, the maintenance of mobile applications remains to be largely undiscovered.” 79, p. 27] The research reported in this dissertation investigates aspects of software maintenance, because it is an essential aspect of the work app developers do for apps they want to support and keep current. Despite their wide-ranging investigations of prior research, none of these publications used or considered mobile analytics.

Following on from the challenges and future directions section on maintenance research for mobile apps: do researchers focus in areas where the streetlights are, rather than where the problems are? i.e. on where they can find material to study rather than on issues that practically affect the majority of developers of apps?

3.4.10 Summary

Despite all the rich and varied research into developing and testing mobile apps, there are gaps in the research in terms of the use, by developers, of mobile analytics. With the exception of Microsoft’s work in mining crash analytics, existing research has not investigated or sought to understand the effects of mobile analytics on being able to measure or improve reliability of the mobile apps in terms of the software development practices, artefacts, or in the mobile analytics tools developers use. And even Microsoft’s insightful research did not discuss what developers did with the information and material that Microsoft provided in their then-current mobile app store.

[44]: https://github.com/SalomonBrys/ANR-WatchDog


[45]: issue-tracker.google.com/issues/36951741

The final area to investigate is what are the sources of information on software quality for the developers of mobile apps. Mobile Analytics could be one such source; it is unlikely to be the only source, so it is useful to place it in the context of other sources.

### 3.5 Sources of information on software quality for developers of mobile apps

Feedback comes in various forms in an app-store ecosystem, ratings and reviews are two of them and possibly the best known ones because they are public and highly visible to end users and anyone else who wishes to see them. As part of scoping this research at least fourteen distinct forms of feedback have been identified, these are presented in Table 3.1.\footnote{As discussed in the previous section, it’s not clear whether the test results of Microsoft’s work on Caipa, VanarSena, and SMASH were ever presented to the developers of the Windows Phone apps Microsoft tested. Nonetheless Microsoft’s approach could be added to this table as and when it has been presented to app developers. As an observation, Microsoft’s work appears similar to pre-launch reports in Google Play – yet significantly more capable.}

The feedback source is mainly from humans for: ratings, reviews, social media, emails to the dev team, manual testing, and code reviews. The feedback from the other sources is mainly automated. Developers need to be involved in setting up many of the feedback sources, and they choose which ones to act on.

Note: the following sources of feedback are not limited to mobile apps within an app store – they can be used for other software; however these other forms of software are outside the scope of this research.

End users can provide feedback through other public channels such as social media (Twitter and Facebook being the largest and best known), or through a variety of private channels ranging from email (Google Play displays the contact email address for each app developer in the Google Play Store) to in-app feedback. Several of these will be mentioned as examples; otherwise they’re also beyond the scope of this research for various reasons.

Feedback can also be generated by software at run time. These include GUI-oriented utilities such as heatmapping tools, application-oriented utilities including logging and mobile analytics, and platform-oriented tools.
## 3.5 Sources of information on software quality for developers of mobile apps

### Table 3.1: Feedback sources about their app for developers

<table>
<thead>
<tr>
<th>Source</th>
<th>App-store ecosystem</th>
<th>User-base scale</th>
<th>Individual-project</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ratings</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Ratings can be provided without a review.</td>
</tr>
<tr>
<td>Reviews</td>
<td>Yes</td>
<td>Yes</td>
<td>Not really</td>
<td>Researched extensively.</td>
</tr>
<tr>
<td>Social Media</td>
<td>No</td>
<td>Yes</td>
<td>Unlikely</td>
<td>N/A for many apps.</td>
</tr>
<tr>
<td>In-app feedback</td>
<td>No</td>
<td>On a per-app basis</td>
<td>Yes</td>
<td>N/A for many apps.</td>
</tr>
<tr>
<td>Email to dev team</td>
<td>Not really</td>
<td>On a per-app basis</td>
<td>Yes</td>
<td>Oft-ignored?</td>
</tr>
<tr>
<td>In-app Mobile Analytics</td>
<td>Varies a</td>
<td>Yes</td>
<td>Yes</td>
<td>Very popular, distinct from the app store, yet close cousins.</td>
</tr>
<tr>
<td>Platform Analytics</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes, for high-volume released apps.</td>
<td>Reports are not available for low volume apps and those with few detected failures.</td>
</tr>
<tr>
<td>Automated unscripted Testing</td>
<td>Only as part of the pre-launch reports</td>
<td>N/A</td>
<td>Yes, generally available free-of-charge</td>
<td>Works at a platform level, e.g., for many apps on that platform without needing in-depth custom scripts b.</td>
</tr>
<tr>
<td>Manual Testing</td>
<td>Only when using test tracks to release the software to the manual testers</td>
<td>Not generally c</td>
<td>Yes, often used to some extent.</td>
<td>Used piecemeal by the majority of devs. Not necessarily done well. Not necessarily done by ‘testers’ or ‘users’, i.e., non-devs.</td>
</tr>
<tr>
<td>Automated scripted tests</td>
<td>No d</td>
<td>No</td>
<td>Yes if devs have created them.</td>
<td>If the development team has created them then they probably also run them as part of the CI/CB mechanisms.</td>
</tr>
<tr>
<td>Heatmaps (Visual Analytics Replay)</td>
<td>No</td>
<td>No e</td>
<td>Yes, developers need to integrate an SDK and use a service.</td>
<td>Appears that they are used by exception only these days.</td>
</tr>
<tr>
<td>Pre-launch reports</td>
<td>Yes</td>
<td>N/A</td>
<td>Yes, for test releases in Google Play.</td>
<td>These include static analysis and automated unscripted testing. They also combine platform analytics.</td>
</tr>
<tr>
<td>Static Analysis Code Quality tools</td>
<td>Only as part of the pre-launch reports</td>
<td>N/A</td>
<td>Yes</td>
<td>Freely available, perceived as noisy.</td>
</tr>
<tr>
<td>Code Reviews</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Oft practiced by teams.</td>
</tr>
</tbody>
</table>

---

a Firebase - somewhat, otherwise less so. Mainly compatible (except F-Droid).
b Some apps, and therefore some tools, support scripts for login to an app, and/or scripts that bootstrap testing to start at the right place in an app. Robo is a good example of this sort of tool. The tools then explore an app using their own internal algorithms.
c Crowd-based testing is covered in various research; AFAIK the largest scale is Huawei testing a fix for ANRs (re: sd-card writes).
d however several connected services offer remote device test labs which can be used for testing new releases.
e unlikely to pass privacy legislation/constraints
3.5.1 Ratings in app stores

Apps with poor ratings are less likely to be downloaded by new users [49]. Ratings also affect where an app appears in search results and whether an app store will choose to promote that app, or another. In the author’s experience, when one team’s Android app’s overall rating dropped from 4.4 to 4.3 stars, the business noticed an almost immediate reduction in revenues from the app in addition to discovering that app was ranked lower in the search results. Therefore ratings are an important measure for app developers, and one they may choose to influence positively.

Gaming ratings includes activities such as an app first asking if a user is happy with the app, and if they say yes then providing a link to encourage the user to rate the app positively in the app store. An illustrative industry case study explained how Novoda improved the rating of a major newspaper’s app using similar techniques [124]. And in terms of the SDKs developers can use for in-app feedback, Apptentive is one of several companies that provides SDKs to help developers optimise their ratings and guides on how to do so within a mobile app [125]. Unsurprisingly, ratings are also a subject of research interest.

Here are several representative papers that focus on engineering aspects of ratings and reviews. AlSubaihin et al. discusses the symbiotic relationship between releases and the ratings and reviews that app received [28, p. 14]. Some developers time their releases based on the feedback they’ve received and monitor ratings and reviews for their latest release to influence the rollout of the new release. There have been innovative ideas on using ratings to prioritise software engineering activities including software testing of Android games apps [126]. And Greenheld, Savarimuthu, and Licorish highlighted the value of developers responding to reviews [40].

In a population of 10,000 Android apps extracted from Google Play, correlations were identified between the density of warnings reported by FindBugs, a static analysis tool, and app store reviews and ratings for these apps [127]. They considered three warning categories: bad practices (which include reports of crashes in the reviews), internationalisation, and performance (which might relate to ANRs, however they didn’t investigate this aspect). They did not investigate whether addressing warnings from FindBugs improved the ratings or reviews for those apps, and mobile analytics was not mentioned in their work – unsurprisingly, as they took a black box approach using decompiled Android app binaries.

Figure 3.6 shares XKCD’s apt summing up two facets of flaws in app store ratings and reviews: averaging reviews may not be a good measure [128], and the star ratings are not treated as a linear scale in practice [129]. In summary, app store ratings therefore may not be an ideal measure of the quality of an app from a software engineering perspective!
3.5 Sources of information on software quality for developers of mobile apps

Similarly, reviews in app stores have been analysed for various purposes including for complaints and bug reports. Of the many and various research studies on reviews of mobile apps in app stores, two early papers are indicative of the literature. The first focuses on Android apps in Google Play Fu et al. [130] and the second on iOS apps in Apple’s App Store Khalid et al. [131].

In Fu et al.[130, p. 5], the three most common indicators of problems in the apps were: slow 9,939, crashes 9,081, and freezes 3,960. Stability was a common theme of complaints about both free and paid apps (ibid.[pp. 7-9]).

In [131], Khalid et al. clearly establish connections between what users of iOS apps complained about and the effects of these complaints on ratings of those apps; and Panichella et al. [132] extend that work to consider the relevance of various review-topics to developers. Panichella et al. also extended the work to include Android apps in the Google Play Store. McIlroy et al. [133] labelled the content of reviews extracted from Google Play and found that crashes and crashing were one of the topics that was a) relatively easy to categorise, and b) occurred frequently. This paper seemed to conflate in-app analytics tools (Flurry) with analytics of reviews (e.g., AppAnnie). N.B., since this paper was published, Google Play now provides automated labeling and analysis of reviews.

One of the curiosities reported by AlSubaihin et al. [28, p. 13] is: “while automatic in-app crash reporting is the most prolific channel of reporting bugs, the one mostly prioritised by our respondents is user reviews in app stores.” Unfortunately, they don’t discuss the effects of the crashes being a lower priority than user reviews; nor do they discuss other sources of crash reports, even though these existed at the time of their research.

Similarly, in terms of managing releases in app stores, the platform-provided release management tools and reports were not mentioned in research. They are not well-publicised or described. For example, they...
are alluded to in a Play Console help article, “In Play Console, you can see an overview of your app’s ratings, individual user reviews and clustered data about your app’s reviews.” [134] without further detail.

Feedback comes in various forms; ratings and reviews are only two of them. One that’s seldom used in industry and seldom researched is implicit feedback recorded through user-interactions with the GUI of the mobile app. Common terms for this mechanism include ‘Heatmaps’ and ‘heatmapping’, and there are commercial SDKs and opensource projects available that perform the recording within an app at runtime. The elixir of automating ‘record and playback’ test automation sometimes uses similar methods to record the interactions with a mobile app.

3.5.3 In-app feedback

Several papers discuss in-app feedback with the aim of helping developers.

The first is a mapping study by Scherr et al. that aimed to identify the requirements for in-app feedback tools to support in-app feedback to improve the quality of apps [135]. After various filtering, they selected 36 papers. They analysed and categorised the types/forms of feedback, e.g., text, audio, video, [file] attachments, and even emotions and behaviour. They also determined whether feedback was initiated by the user or by the app. They created and tested their implementation of in-app feedback using an app designed for citizens to provide feedback to a smart city project. Their test was performed with friends and colleagues of the team, a sample not likely to be representative in terms of the content or behaviours. Nonetheless their research and their prototype in-app feedback may be of interest to app developers in future, especially given their analysis of how to design the feedback.

The paper did not mention software analytics; nonetheless their app might include in-app-software analytics.

Finally, Avellis et al. [1] proposed in-app, bi-directional feedback to help developers and users communicate requirements. This short paper does introduce mobile analytics as a complementary source of information.

3.5.4 Testing

Testing for mobile apps has already been covered in Section 3.4.5. Testing Mobile Apps. Here the focus is on feedback aspects of the testing.

The absence of automated tests does not prove that developers do not test their Android apps; rather it indicates their projects are unlikely to have any automated tests and similarly that the project is unlikely to run any automated scripted tests as part of a continuous build. (In honour of Dijkstra’s observation that “Testing shows the presence, not the absence of bugs”, Edsger W. Dijkstra in [136, p. 16].)

And yet, researchers also continue to complain that app developers do not test their apps sufficiently. They sometimes berate the app developers to “test their apps” [137] (rather than seeking to understand what the
developers are doing and more relevantly why developers do not seem to test their apps to the satisfaction of the researchers.

Another strand is where researchers collate examples of apps together with faults they have identified and analysed in order to help, primarily, other researchers. One of the best-known examples is AndroZoo, a collection of over 3 million Android apps collected from various app stores. The AndroZoo project is described in detail by its authors [138]. Note: access needs to be requested by email 47.

An open question to the research community is: why are all these efforts to improve software testing for mobile apps generating seemingly minimal effects in practice? Perhaps researchers interested in improving mobile apps would find an approach similar to the research of Winter et al. [139] more productive: by actually talking with the app developers and then offering to research pain points identified by those app developers. Doing so might increase the external and ecological validity of the research and potentially also increase the adoption of the research. The adoption might be at an individual developer, team, or organisation level. In some instances the effects of the research may gain wider adoption and become a tool, technique, approach that many app developers adopt. In some cases the app store and/or platform provider might also adopt the work, for instance to replace their current ‘Monkey’ test tool. 48.

There is evidence that developers do want automated tests and tools to provide them with feedback [140, p. 5], even if relatively few projects have sufficient tests to provide the developers with a level of comfort and confidence. Again, much of the research has focused on where researchers can find tests – rather than working with development teams to understand their desires and the barriers that mean the developers don’t have (m)any tests.

For project teams with large volumes of automated tests, what value do those tests provide in terms of feedback about the quality of the app they are intended to test? Fortuitously, one of the app-centric case studies explored in this research, Catrobat, has an opensource codebase, and their uses of automated tests are well researched. The Catrobat project makes extensive use of various forms of automated tests, including using Behaviour-Driven Development (BDD) practices [141], testing under adverse conditions [142], the use of sizing for automated tests [143], etc.

Despite this investigation of testing in the literature, there is limited evidence that provides an understanding of the mobile app developers’ perspectives on testing practices, how tests interact with other information available to developers, and the overall impact on mobile app quality.

3.5.5 Mobile Analytics

In-app analytics have been used to help developers understand ways their mobile app is used by large populations of end-users, and the effects of various conditions on the behaviours of the end users. There has been some research into using mobile analytics to understand usage patterns of specific apps. The first of these papers focused on aspects of usability and the second on helping developers understand the users of their app.

47: androzoo.uni.lu/access

[139]: Winter et al. (2022), ‘Let’s Talk With Developers, Not About Developers: A Review of Automatic Program Repair Research’

48: As an aside Google already has replaced their ‘Monkey’ utility with ‘robo’

[140]: Greiler et al. (2022), ‘An Actionable Framework for Understanding and Improving Developer Experience’

[142]: Adamsen et al. (2015), ‘Systematic execution of Android test suites in adverse conditions’
[143]: Hirsch et al. (2019), ‘An Approach to Test Classification in Big Android Applications’
Using Google Analytics to improve usability testing: Google Analytics for Mobile Applications was used to provide continuous usability logging to address challenges and severe limitations with usability testing in the lab [144]. The authors prescribed a three-stage approach for logging. In the third stage they have two alternatives, one for lab usability testing, the other for continuous usability logging using Google Analytics. The logging mechanism involved writing additional code that recorded various details while the app was being used and then packaging these details in custom events using Google Analytics’s setCustomDimension(...) method. They evaluated their custom logging by collecting and processing the custom data collected via Google Analytics and found the results were sufficiently similar to those obtained in the usability lab to claim that using Google Analytics would be viable for finding usability issues. The authors proposed extending their work beyond usability testing to measuring usability issues for real usage of mobile apps. They described various improvements to their mechanism that might improve the utility of the data collection.

Helping devs understand the users of their app: Automatic instrumentation of mobile apps using mobile analytics tools including HP’s App Pulse Mobile was able to help developers better understand their users [145].

Neither of these publications considered using mobile analytics for other purposes such as measuring the reliability of mobile apps.

Using mobile analytics to improve mobile apps: Research by Patro and colleagues incorporated a toolkit library called Insight into two mobile apps. The toolkit combined passive analytics (such as session length) with factors (such as network condition and client device), with the aim of helping the developers recognise correlations between these factors and application use and revenues [146, p. 82]. Interim aspects of this research were also published by [147] in 2013.

The research describes generic data, collected by Insight for any app, and app-specific data, which is defined and implemented by the developers of that app [146, pp. 87-88]. Patro and colleagues identified a correlation between battery drain and session lengths: as battery drain increased, the session lengths decreased. They also identified a correlation between screen brightness and battery drain. The device model was a material factor in the rate the battery drained, and in particular on Kindle Fire devices the drain was far higher when the screen was bright: “controlling the screen brightness on a Kindle Fire device reduced the average battery drain by 40% while using SB” [146, p. 13] Note: SB is short for StudyBlue, one of the two apps in their study.

The research using Insight was one of the catalytic agents for the research reported in this dissertation, as it was able to publish data obtained using mobile analytics for two popular real-world apps. Patro and colleagues also made both their client [49] and server [50] code freely available online as opensource projects. Although they mention they developed an iOS SDK, it does not appear to have been made available online [146, p. 85].

Perhaps paradoxically, the Insight client SDK does not record or report any failures of the SDK or of the associated app with which it has been
3.5 Sources of information on software quality for developers of mobile apps

integrated. Assuming crashes, freezes, and other similar issues might affect the end-user experience, it seems odd the SDK doesn't track these aspects. Furthermore the SDK only sends the analytics data when there is a working mobile network immediately available; it does not appear to queue events, incorporate any robustness mechanisms, or include any re-transmission capabilities.

The research of Patro and colleagues only provided a brief comparison between Insight and three then-current mobile analytics services (Flurry, AppClix, and Android) [147, p. 12]. Their brief comparison did not go into any detail about the characteristics or use of those three services.

Privacy implications: There has been research into various risks and concerns when using mobile analytics. The following paragraphs present research that provides pertinent advice for app developers, and other stakeholders on how to improve end-users’ privacy when incorporating in-app mobile analytics.

According to research by Zhang et al., 5.2% of app developers of top Android apps in Google Play have violated either their own privacy policy or the terms of service of the mobile analytics service they use in their app [148]. This number may be an underestimate, as the method the researchers used (decompilation of the application’s binary file) did not work on obfuscated binary files, and top (i.e., popular?) apps are likely to be protected through obfuscation. Zhang et al.’s advice to app developers is pertinent: they should abide by both their own privacy policy and by the terms of service of the analytics services they use.

There has also been research into similar sounding work on software analytics for mobile apps: an opensource project called Software Analytics for MOBILE Applications, source samoa.inf.usi.ch/about/ (SAMOA). However, that research investigated characteristics of the source code, rather than in the use of mobile analytics by app developers [149].

Minelli et al.’s Chapter 33 in [50, pp. 177-187] provides an industry perspective on the importance of [mobile] analytics, monitoring, and alerting. This book draws sources from experienced industry practitioners who have worked extensively with mobile apps from an engineering perspective. The chapter explains how metrics are often wrong, yet teams do not notice for years. It then presents the metrics certification process used at Pinterest which helped that company detect flaws with their mobile analytics, often before the analytics went live.

Research into mobile analytics is surprisingly rare, especially given the prevalence of apps that include mobile analytics SDKs and their use by mobile app developers. A possible reason for the rarity is the challenges in researchers obtaining access to the outputs of the mobile analytics. For the research into Insight analytics the researchers developed their own client SDK and server code and they negotiated special non-disclosure agreements (NDAs) and the data was pre-filtered to increase the privacy of the end users [146, p. 91]. This provided them with access to the analytics outputs. Their research did not investigate third-party mobile analytics services or go into details of how developers applied the mobile analytics to effect improvements to the apps or related artefacts.
Privacy aspects: Before we leave the topic of mobile analytics, research into the privacy aspects of the mobile analytics SDKs is pertinent, as the choices developers make in terms of selecting in-app mobile analytics have various consequences. These include consequences for the privacy of the end users who indirectly provide the underlying data. Two illustrative areas of research into the privacy and data leakage aspects are:

▶ A privacy-focused app is used to obtain the network traffic sent to ad and analytics end points from end-user devices. That traffic is analysed to understand where it goes, who has access to the underlying data, and what of the content is most likely to contravene ePrivacy and GDPR directives [150].

▶ In contrast, the binary files of various apps were modified in order to intercept the calls to the respective in-app SDKs of various mobile analytics libraries. They developed a proof-of-concept Android app AlManager that a) allowed the user to see the contents of the calls made to the in-app SDKs, and b) blocked or replaced the contents with blank data. They also studied characteristics of the calls the apps made in terms of the App, Activity, and User level in terms of the data being sent to the mobile analytics SDK(s) [151].

The first of these papers focused on the data that is sent, the contents, and where that data goes. The second concentrated on analysing app binaries to find all the calls to the mobile analytics SDKs. It then intercepted these and enabled users to see, block, or blank out data that would then be sent by the respective mobile analytics SDK.

Finally, in terms of research into mobile analytics, there is a paper that initially appeared relevant as their research was on software analytics for mobile apps. Minelli et al. describe a web-based software analytics platform named SAMOA [149] that was developed to provide insights into the source code of various Android apps. They analysed opensource codebases for apps on F-Droid [55]. While their analysis is of interest [56], they did not investigate mobile analytics. (Also they researched apps on F-Droid; however apps on F-Droid are not permitted to include mobile analytics unless all the SDK and libraries are also opensourced).

3.5.6 Summary

Developers are faced with a plethora of sources of information. Some they choose, such as providing in-app feedback mechanisms and in-app mobile analytics. Others may be forced on them (and testing might be a mix of the two). There is some research into uses of mobile analytics, but little of substance on how it can be used effectively by developers to improve their practices and their apps. Also, there is scant research into the mobile analytics tools and services in terms of the information they provide developers.

3.6 Chapter summary

App stores, their ecosystems, and their effects on developers have been covered from various angles. Some of the research describes snapshots
of one or more app stores. And some of the research involves developers of live apps in a mainstream app store. Ratings and reviews are covered from many angles including fraud [152]. Yet few have investigated the use of mobile analytics.

Following on from a point raised earlier in this chapter: “between detecting failures and achieving reliability.” [54, p. 77], mobile Analytics might be able to provide a real-world measure in terms of achieving reliability. Furthermore, much of the research into mobile app crashes seems to end prematurely – before considering whether reliability of the app has been improved by whatever method and approach they have researched.

“...with explicit and implicit feedback now available (almost) continuously, questions arise. How can practitioners use this information and integrate it into their development processes to decide when to release updates?” [34, pp. 48-49]

Building on a point made by Maalej et al.: "the future of app quality engineering is data driven.” [79, p. 24], mobile analytics incorporates rich seams of data that may help developers perform app quality engineering. However, in order to help developers mine this data more effectively and process it into actionable information, we need a better understanding of current practices associated with using mobile analytics. This includes investigating how mobile analytics use is integrated into mobile development practices, and the effects it has on the quality of the software artefacts associated with mobile apps. Further, we also need to understand the capabilities and limitations of existing mobile analytics tools. These gaps in our understanding lead to the six perspectives introduced in Chapter 1. Introduction and the overarching research question for this thesis: How can applying mobile analytics in software development practice improve the reliability of mobile apps? (Reproduced from Section 1.3. Research Questions.)

The next chapter explains the methodology used in this research to identify and record ways developers use mobile analytics currently, the sources of pertinent data, and the approach this research took to answer the research questions.

[54]: Frankl et al. (1997), ‘Choosing a testing method to deliver reliability’
[34]: Maalej et al. (2016), ‘Toward Data-Driven Requirements Engineering’
The research was designed to address the main research question: *How can applying analytics improve software development and software testing for mobile apps in practice?*. The six perspectives – introduced in the Introduction (as part of the Section 1.3. Research Questions) and repeated here in Figure 4.1 for ease of reference – direct attention to two key dimensions of investigation:

1. **the different objects of analysis**: the software development processes used by app developers, the software product (*i.e.*, app) and related development artefacts, and the mobile analytics tools;
2. **a temporal dimension** considering what *is* (*i.e.*, the current status) and what *could be* (*i.e.*, scope for improvements relating to the objects of analysis.

The research is both knowledge-seeking and solution-seeking in nature [153, p. 11-4]. It is knowledge-seeking in terms of understanding the use of mobile analytics in the practices, development artefacts, and in understanding the mobile analytics tools that are being used. It is solution-seeking in terms of considering improvements to each of these three objects of analysis.

---

**Figure 4.1: Methodology six perspectives (repeated)**
4.1 Evidence requirements

Answering the research question requires rich, contextualised evidence of how developers use mobile analytics in practice. Hutchins writes about studying ‘cognition in the wild’ to provide rich and emergent findings from the real world for phenomena that are hard to capture and analyse without their rich context [154]. Hence, the research needs to be situated in real-world industry practice and experience, drawing on different examples of real-world apps and projects for which analytics have relevance, i.e., that are part of an app-store ecosystem that collects analytics, and that are used by real-world users.

A comprehensive statistically representative sample of mobile apps was beyond the scope (and feasibility) of a PhD. Instead, this research adopts the approach of ‘purposive sampling’ as described by Flick [155, pp. 180-182]. A similar approach has been applied successfully by other researchers undertaking in-depth, qualitative explorations of software development practice, e.g., recent work on qualitative analysis of pair programming undertaken by Zieris, who observed: “Unlike for quantitative methods, statistical generalization from a sample to the population is not a goal. Instead, qualitative studies look for information-rich cases that allow to deepen the researcher’s understanding. Early in the process, each case is treated as unique and studied in great detail; cross-case analyses follow later and are based on the well-understood individual cases.” [156, p.114].

Flick presents six strategies of purposive sampling [p. 181]. Of necessity the sample used by this research was opportunistic – what Flick described as “convenience” sampling. As Flick notes wryly: “the problem of access may be one of the crucial barriers” [p. 182], which applies particularly when seeking access to sensitive data and information about software failures for commercial mobile apps. Nevertheless, the research strove to use a variety of projects and apps including: commercial and volunteer-led development teams; solo developer; small, medium and large development teams; industry and opensource apps; apps that include in-app mobile analytics and those that choose not to. Some of the projects that declined to participate in the research would have helped address some of the gaps in coverage of the known varieties of projects and apps.

Further, the research explored a variety of analytics tools, as no two are identical: they offer a variety of features and capabilities, and have distinct behaviours. Furthermore there are tools that work at the platform level and others that work at the app level; researching tools that work at both levels helps determine and distinguish their characteristics and compare their behaviours. There is only one platform-level mobile analytics tool in Google Play, the one that Google provides. In contrast, there are tens of app-level mobile analytics tools, so there is value in studying several these app-level mobile analytics tools.

Case studies were used extensively during the research as vessels for engaging with both the app development teams and the mobile analytics development teams. The choice of case studies is intended to provide variety and the opportunity for comparison and to provide a multi-layered approach.

The varieties of projects, apps, and mobile analytics tools, are all intended to help to uncover emergent features, capabilities, and behaviours; they
also help establish ranges of examples of improvements and concerns. An additional objective is to increase the ‘weight of evidence’ in support of particular propositions, rather than to prove them [157, see p. 569], which is impractical in the scope of this research.

4.2 Data sources

The nature of research in a sometimes-messy real-world environment means access is opportunistic and often occasional. The evidence will be incomplete, yet rich, complex, and multi-faceted because of the variety of projects investigated. For this research, the data sources include:

- **Development artefacts**: Their origin is the development team. They include: app binaries, app source code, tests, work schedules, documentation, bug tracking systems, etc. These were collected during the various case studies and complemented by public sources for additional mobile apps.

- **Grey material**: Some examples were gathered during the case studies; others were found during additional background research.
  - Grey data includes: various discussion forums used by mobile app developers and other online contextual information, online issue tracking and related code for opensource projects beyond those that were the focus of the app-centric case studies (e.g., open source networking libraries used by Android apps).
  - Grey literature includes: online materials on mobile analytics tools, articles including material published on medium.com and in various blogs. Some were found in response to observations during particular case studies; others were found during additional background research.

- **Pre-study interviews**: with developers and, as appropriate, authorised representatives of their organisation. These were used for setting up the study and understanding the development context. They were collected as part of the case studies.

- **Mid-study communications with developers**: usually email correspondence for clarification, to obtain updates, or comment on observations. These were collected as part of the case studies.

- **Field notes**: some handwritten, others recorded as text on computers. These were collected as part of the case studies and during background research.

- **Analytics tools and associated analytics artefacts**: their origin is the mobile analytics tool. Analytics artefacts, in particular, were a key data source; they included various outputs including screen captures, screen-scraping and parsing, results from calling APIs, and automated emails generated by analytics tools. Product documentation, online help materials, examples, and so on were also used. For open source analytics tools, the code was also used as a data source. All these data sources were collected on an ongoing basis during the case studies and during additional background research.
The evidence is based mainly in real-world cases, augmented with micro-
experiments where these were appropriate. For all of these real-world
cases, collection of naturally-occurring data (e.g., development artefacts,
grey material) was augmented by elicitation of additional data (e.g.,
interviews and communications with developers) to provide clarification,
breadth and insight. Different research methods made use of these data
sources, as appropriate.

In terms of the methodology, during the case study it is vital to collect
and perform ongoing analysis of mobile analytics and whatever other
materials are available. Many of these are ephemeral in nature. For in-
stance, graphs generated by analytics tools may change by the minute.
There is seldom a manual for the mobile analytics outputs (e.g., the
reports). Furthermore, many of the reports are dependent on the under-
lying data and/or on changes in the underlying service; therefore the
researcher often needs to learn the mobile analytics reporting iteratively
in an exploratory manner. For example, one of the findings, reported in
Chapter 8 in the topic Runtime encapsulation of failures, was the runtime
encapsulation of crashes in the application code. This was discovered
through one of the app-centric case studies and corroborated by a second
one, and then also through grey data. They were not documented a
priori.

Third-party mobile analytics (including those provided by Google) have
terms of use. These terms of use have various names, such as a ‘policy’,
e.g., for Google Play [158]. These may place limitations on data collection
and use of the relevant service. For the research covered in this thesis,
data collection used a conservative approach to reduce the risk of conse-
quential issues for the researcher, the project, and the stakeholders for
the app.

The choice of tools, including the humble web browser used by the
researcher, affects aspects of the ease of collection of online reports. As
an example, the screenshot capability of the Mozilla Firefox browser ¹ is
far richer than that provided by Google Chrome at the time of writing.
Many of the reports in mobile analytics tools require extensive vertical
scrolling; Firefox can capture the entire contents easily, but Chrome does
not.

Similarly some content is only generated on screen on demand, in
response to user actions, for example by scrolling vertically (such as
using ‘infinite scrolling’ [159]) and/or paging through reports. Others
are contextual and may only appear when the relevant conditions occur.
For example, the release management reports in Google Play Console
appear for the first 7 days of a new release.

Therefore, to capture the content, the researcher (or a human/automated
proxy) needs to perform these actions and to save/safeguard pertinent
materials to facilitate longer term analysis and provide/record evidence.
Where practical, the underlying text was collected in addition to visual
content; we created software called Vitals Scraper do so for Google Play
Console with Android Vitals. The text could then be processed relatively
easily and without needing to be re-keyed. Note: it is not always practical
or useful to record “everything”; how much is suitable is a topic for future
research.

¹: Described in https://screenshots.
firefox.com/

[158]: Google (2019), Print view: Google
Play Developer Policy Center

[159]: Parker et al. (2012), Infinite Scrolling
4.2 Data sources

Table 4.1: Mapping data source (rows) to the 6 perspectives (columns)
S = Strong contribution
m = moderate contribution

<table>
<thead>
<tr>
<th>Six perspectives</th>
<th>Understand</th>
<th>Improve</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>use</td>
<td>artefacts</td>
</tr>
<tr>
<td>Development artefacts</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>Grey Data</td>
<td>S</td>
<td>m</td>
</tr>
<tr>
<td>Grey Literature</td>
<td>m</td>
<td>m</td>
</tr>
<tr>
<td>Pre-study interviews</td>
<td>S</td>
<td>m</td>
</tr>
<tr>
<td>Mid-study communications with developers</td>
<td>m</td>
<td>m</td>
</tr>
<tr>
<td>Field notes</td>
<td>S</td>
<td>m</td>
</tr>
<tr>
<td>Analytics tools &amp; their artefacts</td>
<td>m</td>
<td>m</td>
</tr>
</tbody>
</table>

4.2.1 Mapping the data sources to the six perspectives

The various data sources described in the previous section each contribute to the six perspectives as indicated in Table 4.1. The table has two indications of the strength of the contribution: ‘S’ indicates a strong contribution, and ‘m’ indicates a moderate contribution. The blank cells indicate low, or marginal, contributions rather than necessarily no contribution at all.

Development artefacts contribute strongly to understanding of the use of mobile analytics and the artefacts themselves. They also contribute to the understanding of mobile analytics tools, for instance as they contain the usage of any API’s provided by a mobile analytics tool. Through understanding the development artefacts, various potential improvements emerge for each of the three objects of analysis.

Grey data, e.g., developer-centric discussions in project issues on GitHub and Q & A on StackOverflow, contribute mainly to understanding the current state of affairs, both the immediate issues and historical issues that may or may not have been addressed or retired through changes to the mobile analytics tools and any of their associated SDKs. While they may also hint at future improvements, that’s seldom the focus of the developer-centric discussions, although occasionally external developers may also suggest and/or contribute specific improvements.

Grey literature can contribute moderately to all six perspectives, limited partly because the material tends to be general in nature rather than specific to particular apps or projects.

Pre-study interviews contribute mainly to understanding a team’s current use of mobile analytics tools. They seldom get into detail about the development artefacts, apart from various statistics that are reported by mobile analytics tools, e.g., the crash rate of the team’s app(s). They often indicate areas of improvement across the board that the team could make, but not in enough detail to provide concrete improvements at this early stage in the relationship.

Mid-study communications are often focused on understanding immediate and recent events from a variety of sources (including use, changes to the artefacts, outputs from the mobile analytics). During discussions that are part of the communications, scope for improvements emerge across the board.
Field notes may well be the strongest data source overall; the only area where they’re limited to a moderate contribution is in terms of understanding the artefacts – generally the understanding of the artefacts is evidenced primarily in the development artefacts directly, so the field notes augment these rather than being the strongest source of information for artefacts.

Perhaps unsurprisingly, analytics tools and associated artefacts contribute most strongly to the use and improvement of the tools themselves. They also contribute strongly to improvements in the artefacts, e.g., through identifying areas in the source code that lead to a crash.

4.2.2 Interviews

Interviews were one of the key data collection methods. They were qualitative in nature and, as such, primarily open and flexible in their arrangements, in order to facilitate interviewees to provide rich experiences and insights as they arose. The case studies included more senior participants and volunteers; these interviewees have a great deal of autonomy and choose what they are willing to provide, including whether they respond at all, (see [160] for a discussion on these aspects). And all participants ultimately had a great deal of freedom in practice in terms of their contributions. Hence, interviews were open and flexible, even when the protocols were prepared in advance.

The nature of open and flexible interviews with senior and voluntary participants meant that the interviewer needed to be artful in the design and conduct of the interviews, in order to maximise the potential value of those interviews. At times, in an effective interview, questions may alternate between the interviewer and the interviewee [161].

The interview questions always covered at least these two fundamental topics in addition to any additional topics: 1) access to the tools in order to conduct the research (research activity), and 2) primary data (questions pertaining to their use of mobile analytics).

Note: many of the case studies overlapped with various lockdowns related to coronavirus, when interviews in person were impractical. That said, email-based interviews were already recognised as an established research practice [162]. Similarly, online interviews, (c.f. [163]) using various video-conferencing services, formed a key aspect of the research.

The data from interviews was analysed using the sense-making and sense-building methods described in the next section.

4.3 Methodology

The methodology builds on three primary complementary sources: Ball and Ormerod’s ‘cognitive ethnography’ [164], Runeson and Höst’s ‘guidelines for conducting and reporting case study research in software engineering’ [165], and Seaman’s ‘Qualitative methods in empirical studies of software engineering’ [157] 2.
Broadly, this research starts from what Ball and Ormerod described as ‘cognitive ethnography’ [164], that is, observation-based enquiry conducted *in situ* to investigate “...the interplay between people-laden contexts and expert cognition” [p. 149]. Ball and Ormerod characterised cognitive ethnography in terms of observational specificity, verifiability and purposiveness:

“Our own conception of cognitive ethnography is characterized by three key features. First, it relies on small-scale data collection based around representative time slices of situated activity. As such, it demonstrates observational specificity, as opposed to the intensity of a prototypical ethnography. Second, it is purposive, in that its mode of questioning focuses on issues that are informed by some intention to intervene with, or somehow affect, existing work practices ... Third, it places a strong emphasis on verifiability, in terms of validating observations across observers, data sets and methodologies.” [164, p.152]

Consistent with this orientation, this research sought to derive a situated understanding of analytics (in terms of the six perspectives in Figure 4.1). The insights that emerged from the more ethnographically-inspired analysis of naturally-occurring data were then investigated further and tested using other approaches, namely across-case comparison, hypothesis testing (systematic manipulation in micro experiments), and action research (evaluation through interventions in specific cases) – consistent with Ball and Ormerod’s emphasis on verifiability.

### 4.3.1 Case study methodology

Case studies were the primary mechanism in this research. As Runeson and Höst note, software engineering motivates specialised research methodologies, as: 1) the study objects are organisations that *develop* software, 2) the work being studied is project-oriented, and 3) that work includes advanced engineering work by highly educated people [165, pp. 132-133].

Case studies have been established by various researchers as an effective method to understand software qualities, for instance in various research by Khalid et al., in (2014, 2016, 2015), and the work of [166]. And case studies are well suited to exploratory research, for which they may offer insights not available with other approaches [167], especially when events are contemporary and where the investigator has little or no control in [168, Chapter 1]. They are also recommended where the research includes multiple data sources [167]. This research therefore uses case studies, as the research investigates real-world use of mobile analytics where many factors are outside the control of the researcher.

These criteria apply to all the case studies in this research, and therefore a variety of research methods was adopted in order to perform the research effectively and productively.

This research is situated in development teams that create mobile apps and use software tools – in particular mobile analytics – in order to provide apps that work adequately for their user bases. As Seaman notes, it’s important to study “nontechnical issues and the intersection between the
Methodology

4.3.2 Categories of research methods

Figure 4.2, on page 76, provides a visual overview of the four categories of research methods used to serve this ‘cognitive ethnography’ approach. They include: 1) sense-making, 2) sense-building, and 3) evaluation through action research, which were complemented by the fourth category 4) feedback mechanisms to help support and verify the analyses.

Note: Figure 4.2 is an over-simplification with clear boundaries between the categories in order to convey the alignment of methods to purposes, and the categories are not discrete. Some methods include several data sources, and some data sources are analysed using several methods. In practice, individual research methods provided multi-faceted contributions; for instance local app experiments contributed to both sense-building and sense-making.

Table 4.2 identifies the various research methods, groups them in terms of their roles in the research (as illustrated in Figure 4.2), and maps them to the six perspectives. Each of these research methods includes data collection and analysis unless otherwise stated.
Table 4.2: Mapping research methods (rows) to the 6 perspectives (columns)
S = Strong contribution
m = moderate contribution
FOSS = Free and Open Source Software

<table>
<thead>
<tr>
<th></th>
<th>Understand</th>
<th>Improve</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>U_{use}</td>
<td>U_{artefacts}</td>
</tr>
<tr>
<td><strong>Sense-making</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(comprehension and exploration)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Beacon finding and</td>
<td>S</td>
<td>m</td>
</tr>
<tr>
<td>Drill-down</td>
<td>m</td>
<td>m</td>
</tr>
<tr>
<td><strong>Sense-building</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(micro-experiments and macro-discoveries)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Local App Experiments</td>
<td>m</td>
<td>S</td>
</tr>
<tr>
<td>FOSS Analytics Experiments</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>Across Case Comparisons</td>
<td>S</td>
<td>m</td>
</tr>
<tr>
<td><strong>Feedback mechanisms</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(triangulation and validation)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ask The App Devs</td>
<td>S</td>
<td>m</td>
</tr>
<tr>
<td>Ask The Tool Devs</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>Grey Literature &amp; Grey Data Analysis</td>
<td>m</td>
<td>m</td>
</tr>
<tr>
<td>Code, &amp; app-binary, Analysis</td>
<td>m</td>
<td>S</td>
</tr>
<tr>
<td><strong>Action research</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(embedded intervention)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Observation and Analysis</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>Field Experiment</td>
<td>m</td>
<td>S</td>
</tr>
<tr>
<td>Hackathon</td>
<td>m</td>
<td>m</td>
</tr>
</tbody>
</table>

‘**Sense-making**’ methods were concerned with understanding current practice (as reflected in artefacts, tools, and developers’ practices and perspectives – i.e., perspectives U_{artefacts}, U_{tools}, and U_{use}) and identifying potential improvements in tools and in how analytics are used in app development and maintenance (i.e., perspectives I_{tools} and I_{artefacts}). The research methods include beacon-finding (see page 78) and drill-down (see page 80).

Sense-making incorporates an iterative pattern of **beacon finding** to identify areas of interest within a case, **drill-down** to investigate one or more beacons, and comparisons both within and across cases to identify patterns, relationships, and counterfactuals, as some characteristics emerge in contrasts across and between studies. Comparisons were performed iteratively on an ongoing basis (for active apps, the values are not constant). These comparisons included comparisons across releases, using different windows of time, on different dates, comparisons with peers, and so on.

‘**Sense-building**’ methods build on – and test – insights found through sense-making. The research methods included micro-experiments, carried out on local apps (see page 81), FOSS analytics experiments (see page 82), and across-case comparisons (see page 82).

Local app experiments (see page 81) were used to investigate detail and give insight into the relationships between tools, quality of analytics, and potential impact of analytics use on apps (i.e., perspectives U_{artefacts}, U_{tools}, I_{artefacts}, I_{tools}).

Across-case comparisons (see page 82) identify macro-discoveries – that is, they identify characteristics and patterns that were not evident in individual case studies, potential improvements to practice (i.e., perspectives

3: While across-case comparisons are grouped under sense-building, they also help with sense-making.
4: “…relating to or expressing what has not happened or is not the case.” Oxford Languages.
4 Methodology

U\text{use} and I\text{use}, and the influence of the quality of tools in practice (I\text{tools}). They can also corroborate and/or challenge findings found in individual cases.

‘Feedback mechanisms’ were used to support and verify the other analyses, by comparing observations to other evidence, or by asking developers for clarifications or reflections. Feedback mechanisms were used throughout the research and contributed mainly to the understanding of perspectives associated with (U\text{use}, U\text{artefacts}, and I\text{tools}); nonetheless they were also used frequently when considering improvements (I\text{use}, I\text{artefacts}, and I\text{tools}).

‘Action research’ methods were concerned largely with understanding the use of analytics in context and the evaluation of the effect of improvements in the use of analytics, in terms of adoption into use and app performance (i.e., perspectives I\text{use} and I\text{artefacts}). It includes three research methods: 1) observation and analysis, 2) field experiment, and 3) hackathon. These are explained on page 87 onward.

The research was iterative, moving through sense-making, sense-building and feedback mechanisms repeatedly as new cases were studied or new insights emerged. The methods and the data sources often also informed several of the six perspectives. These research methods are introduced in more detail in the sections that follow.

4.3.3 Sense-making

‘Sense-making’ focuses on understanding the current practices and identifying potential improvements in practices and tools. It includes beacon finding (inductive analysis of different forms of data to identify areas of interest) and drilling down (further data collection and analysis to understand those areas of interest in context). Sense-making can include other activities, such as:

- Collating similar failures.
- Ordering and ranking clusters of failures.
- Bug identification and localisation: Establishing potentially pertinent patterns in the reports, and characterising when a failure does and does not occur are part of this work. Obtaining an identifying definitive boundaries may be impractical; the work is often iterative and exploratory in nature and lossy.
- Bug investigation.
- Checking whether there is likely to be sufficient evidence for any triage process.

Beacon-finding

The notion of ‘beacons’ is borrowed from research on program comprehension; for instance by Wiedenbeck, who helped establish the notion of beacons in software development: “In programming, beacons are lines of code which serve as typical indicators of a particular structure or operation” [169,
The notion of beacons was generalised for this research to include indicators in the analytics of something of interest, or something that required attention. And so ‘beacon finding’ was an inductive process by which significant indicators in the analytics data were identified – and then used to identify areas of the code base that required further investigation. Mobile analytics may include source code that calls APIs in the application’s source code; however the bulk of the contents that need to be comprehended comes as outputs from the mobile analytics tools, and therefore the beacons will differ accordingly. In this research, the beacons include: the shape of graphs in mobile analytics reports, failure clusters, a method call in stack traces, and so on.

*Note: In this thesis text in blue boxes are to provide some additional relevant information.*

**Beacons in Mobile Analytics**

In this research the beacons include: the shape of graphs in mobile analytics report, the contents of one or more failure clusters, a method call in stack traces, and so on.

Candidates for beacons include (numbered for ease of reference):

1. An aggregate increase in error rate.
2. Early adverse trends during a phased release rollout.
3. Correlations with one to a few predetermined factors (e.g. OS release, device model, ...).

The beacons were spotted through a mix of pattern recognition, visual anomalies, adverse trends, *etc.* Often they were grounded through experience of being familiar with working with the artefacts of mobile apps.

Beacons emerge in various ways. For instance in reports they include: anomalies within a report, mismatches and inconsistencies between two sibling reports or between a master report and the linked detailed report.

Inspirations for this work include: Scott Barber’s work on modelling software performance [*testing*] [171].
anomalies within a report, mismatches and inconsistencies between two sibling reports or between a master report and the linked detailed report, in the shape of the curve of a graph, in the distribution and groupings of aggregate data, and so on. Similarly, alerts as determined by a mobile analytics service may be considered potential beacons being ‘promoted’ by the mobile analytics reports.

The most common method of recording potential beacons in this research was using web browser screenshots and/or other mechanisms that preserved information electronically. Some were annotated as part of the beacon-finding and drill-down analysis. Additional notes were written both electronically and/or in physical notebooks.

Selection criteria included: top ranking results, atypical rates of change, adverse changes to reliability, novel failures particularly for the most recent release, etc. A consistent and overriding criterion was to seek beacons that indicated flaws and failures that could materially and adversely affect the user experience of the app for one or more users of the app.

**Drill-down**

Beacons identify areas of interest; these need to be investigated further by ‘drilling down’ and examining the underlying information sources. For example: Is the identified beacon genuinely significant? Does it generalise to other contexts? What does it signify or relate to in the app and its usage, etc? Drill-down starts with the original data sources, but may draw in other data sources to clarify relationships, responses by the developers, etc.

For Quadrant 1 of Figure 4.3 (i.e., the same app and the same mobile analytics tool) examples include:

- Going into greater detail in the current report for a ‘failure cluster’ to understand its characteristics.
- Investigating in more detail in order to understand the report, the nature of the problem, the causes, and effects.

For Quadrant 3 (i.e., a different app using the same analytics tool) examples include:

- Looking across apps to see if the same and/or similar failures were happening in any of those other apps.
- Looking at the characteristics of the failures in the various apps, for instance: does a failure happen as often? on the same release of the operating system? on the same device models? and so on.

For Quadrant 2 (i.e., the same app using other analytics tools):

- Does the same failure appear in all the analytics tools, and if so, are the characteristics similar, or do they differ in particular ways?

Quadrant 4 (different apps and different tools) did occur sometimes in the case studies, however these examples were beyond the scope of this research, as this research focused on Android apps and the other apps were not Android apps; e.g., one of the different apps was a web app.
The drill-down may be expanded further using feedback mechanisms, e.g.:

- Searching grey data and grey literature and cross-referencing of materials.
- Asking people: for instance colleagues, the developers of the app, the developers of the mobile analytics tool, etc.
- Check development artefacts, as these may provide relevant and pertinent information and clues.

A representative example of using the feedback mechanisms is:

- A new release of one of the apps in case study C1 resulted in a five-fold spike in the crash rate for the new release. After drilling-down into both the platform and the in-app analytics, the source code of the network library was reviewed in tandem with searching for similar issues reported on StackOverflow. These helped with identifying the epicentre of the crashes.
- Automated tests for the network library were then used as a basis for devising similar tests for the app. These tests reproduced the crash and demonstrated the subsequent changes to the app were effective in fixing the crash.
- And when the modified app was released, the mobile analytics confirmed that the spike had been addressed successfully in the new release.

This example served both the research needs and the practical project needs; and it transpires that developers use a similar form of sense-making; this is discussed in Section 6.2.2. Sense-making and decision-taking by developers, starting on page 140.

4.3.4 Sense-building

Sense-building moves beyond sense-making. Sense-making aims to understand what is, while sense-building extends sense-making with direct action and more active research, for example: to devise and run experiments to learn more about the behaviours of mobile analytics, and to seek patterns and generalisations across tools, and case studies.

The main methods used for sense-building were local-app experiments, experimentation with Free and Open Source Software (FOSS) analytics tools, and across-case comparisons.

Local-app experiments

Local app experiments were used to test the understanding of the relationships between the usage and the analytics through manipulations of apps and observation of the effects – hence, the local app experiments were used to test some of the patterns identified in the inductive analysis. They consist of micro-experiments, which involved creating and developing small mobile apps intended to exercise particular aspects of mobile analytics. The inputs to an app were directed, in order to determine the outputs from mobile analytics. This was essentially a form of black-box test, where the analytics tool being studied was the System under test (SUT). These micro-experiments helped to address questions and gaps...
observed as part of sense-making. In particular, they provided insights into the relationships between tools, quality of analytics, and potential impact of analytics use on apps (i.e., perspectives $U_{\text{artefacts}}$, $U_{\text{tools}}$, $I_{\text{artefacts}}$, $I_{\text{tools}}$).

The apps that were developed as local-app experiments are small mobile apps and not intended for production use. They were developed in order to investigate the relationships between the inputs (such as usage) and the analytics outputs (such as reports) through deliberate manipulation of the inputs and observation of the consequences.

The local-app experiments allow for tighter control on the usage, (i.e., the inputs) in comparison to the unfettered use of real-world mobile apps. They can help surface behaviours and provide for tighter evaluation in the early, pre-launch/pre-production phases. The local app experiments were used to test some of the patterns identified in the inductive ‘sensemaking’ analysis.

Where practical, the source code of these apps is made available under a permissive open source license to facilitate further research by others and in order to facilitate inspection by and feedback from others.

The results of the local app experiments helped augment the across-case experiments, particularly in terms of providing additional results to compare to those from the app case studies. They also informed discussions with the app and tool developers.

**FOSS analytics experiments**

The research included experiments on several FOSS analytics tools to gather evidence on how these tools worked. Additionally, these experiments explored the processes used by the relevant FOSS projects to accept and adopt contributions that aimed to improve one or more elements of the respective codebase, and thereby improve the respective analytics product offering.

As the research had access to the source code of the tools studied, the investigation of how these analytics tools worked can be considered to be a form of white-box testing, where simple apps were used as test drivers to exercise the functionality of the analytics components. The exploration of the project processes for adopting changes was more like grey-box tests, as there was only partial visibility of the dynamics of the relevant project teams. The FOSS analytics experiments were relatively small in scope and were conducted on an ad-hoc rather than a systematic basis. They were intended to be exploratory in nature to deepen understanding of how willing the owners would be to receiving contributions to their products.

**Across-case comparisons**

Across-case comparisons were concerned largely with understanding current practice and identifying potential improvements to practice (i.e., perspectives $U_{\text{use}}$ and $P_{\text{use}}$), as well as the influence of the quality of tools in practice ($I_{\text{tools}}$). This method draws on the approach discussed in [157, pp. 567-569], e.g., to compare pairs of case studies to determine
variations and similarities, with the additional considerations of seeking areas of potential improvements based on findings from the various case studies.

Similar to using various software testing techniques to find bugs, making comparisons across the case studies can help to identify more of the behaviours of mobile analytics, their use, and their efficacy. Across-case comparisons can increase the probability of seeing fresh characteristics and establish similarities across cases. The comparisons across case studies help establish norms (which can also be used to identify beacons, and anomalies), patterns (and anti-patterns), variety, and ranges.

Comparison across cases (i.e., across projects and their mobile apps) helps both researchers and developers, albeit their interests and focus may differ.

- **For developers**: they can compare the reports for their apps with the results others obtain. Doing so may help them identify problems-in-common (shared problems) and fixes-in-common that work for many apps with similar failures. They can also establish norms, and the comparisons help provide triangulation and perspective. Some developers also find peer-group comparisons stimulating.

- **For researchers**: across-case comparisons include ‘plus one’ [172, pp. 28-29] research that can help uncover emergent behaviour, reinforce existing findings, and so on. The across-case comparisons also provide additional microcosms where new findings are discovered in the behaviours of apps, the tools, the development practices, and the efficacy of the use of mobile analytics performed by a wider variety of development teams.

The data and the system state for individual apps can help to increase the ‘coverage criteria’ for a mobile analytics tool (which may be considered a system or simply a ‘box’ as in black-box, grey-box, or white-box system).

We know from a related domain, software testing, that more test cases can increase the detection rate of behaviours. For example, a paper by Briand [173] discusses concepts including the concept of a cost-effectiveness curve and of what the author terms ‘random variations’, and the effect these random variations have on fault detection effectiveness [p. 5]. The set of inputs (e.g., crash reports) into a mobile analytics tool may result in various beacons – and related data – appearing in the outputs. By increasing the sets of inputs, particularly from dissimilar apps, there is the potential to increase the appearance of beacons. And the larger volume of beacons across the projects allows for weightier analysis.

A key observation is that detection is not static, nor a one-off value; behaviours come and go, particularly in mobile analytics tools where reports are often ephemeral. The value of across-case comparisons increases when the sampling increases to record more of the ephemeral behaviours and outputs.

The nature of this PhD research, based on a relatively small number of apps and associated case studies, means it is premature to attempt to systematically plot the number of cases against the behaviours that were observed. Instead, the focus is on monitoring new insights vs. resonance, and how often a new case highlights something that was not noticed in previous cases, even though it was there.
4.3.5 Feedback mechanisms

Sense-making and sense-building methods are centred on the research and driven by the researcher’s focus and perspective. Unaided – even if they are productive – they risk being marginalised and disconnected from the work of others, in particular the work of those who actually develop the apps and the tools.

This research used four ‘feedback mechanisms’ to validate and challenge the research outputs of sense-making and sense-building:

▶ ask the app developers,
▶ ask the tool developers,
▶ analysis of grey literature and grey data,
▶ analysis of code.

The first two (i.e., the mid-study communications with developers) extend understanding of topics emerging from sense-making or sense-building, consistent with Ball and Ormerod’s ‘cognitive ethnography’ [164] and Petre’s ‘targeted observation’ [175, p.234], by engaging with the developers who are situated in their actual practice in their real-world microcosm. For example, the developers can be asked about their experiences, perceptions or reasoning about emergent observations, or to explain particular decisions or practices that have been highlighted by the research. The latter two feedback mechanisms provide comparison to additional data drawn from analysis of existing information (i.e., grey literature, grey data, code). The feedback mechanisms draw on multiple data sources, including some which are independent of the research, hence allowing for comparison/triangulation/colligation – all with a focus on the research questions.

Ask the app devs

The developers of the apps are uniquely able to voice their perceptions and thinking on their use, experience, and perceptions of using mobile analytics. They are therefore well-placed to provide feedback on observations about their use of mobile analytics and also answer questions about their use of mobile analytics in their development artefacts and their perceptions of the mobile analytics tools.

They are also busy with the challenges related to developing and improving the mobile apps for which they are responsible. As Petre notes in [175] experts are willing to try/explore many tools; however they focus on what can help them, and they may discount many aspects of the mobile analytics tools, including some of the characteristics and behaviours of interest from a research perspective. Nonetheless, from a research perspective, it may be useful to learn more about why they have discounted or rejected various aspects of the tools.
4.3 Methodology

Ask the tool devs

The tool developers understand their mobile analytics tools in depth, and many have a unique vantage point from which to observe how their mobile analytics behave across a large population of apps. Furthermore they are well placed to design, implement, and release fixes and improvements to their mobile analytics products and services. They understand the rationale of their mobile analytics tools and their user-base. And yet, they do not know everything about how their tools are used or perceived, and many are keen to receive feedback and insights accordingly.

The research method entails communicating with knowledgeable, available, and interested people who develop (in the broad sense) the relevant mobile analytics tool. The communication may be direct or indirect (for instance via their customer service, via online feedback links, or in the form of contributions to their opensource project). Being able to provide succinct, clear, timely, and relevant communications may increase the chances of engaging in a mutually productive dialogue.

Grey literature and grey data analysis

A great deal of grey literature and grey data [176, pp 219-221] is available online, covering mobile analytics and to errors, problems related to source code and libraries used in real-world mobile apps. The main sources of relevant grey data include Stack Exchange websites frequented by mobile app developers (particularly stackoverflow.com) and GitHub.

Both Stack Exchange and GitHub provide comprehensive and useful search capabilities which help find relevant content, for instance examples where other development teams have found, understood, and addressed particular crashes in their Android app that also apply to the app in the case study.

stackoverflow.com provides facilities that encourage meta-data to be provided by users of the site, e.g., tags, votes, accepted answer flag, etc., and their search provides facilities to perform searches that use meta-data as well as free text [177].

The nature of GitHub projects provides some inherent structure which can be utilised when performing searches, for example to search issues and pull requests ⁶.


[177]: Stack Overflow online help (2021), How do I search? Help Center, Stack Overflow

[178]: GitHub Inc. (2021), Searching code - GitHub Docs

[179]: Scaffidi et al. (2007), ‘Toward a Calculus of Confidence’

[180]: Scaffidi et al. (2007), ‘Developing confidence in software through credentials and low-ceremony evidence’
and answers on StackExchange sites and on public issue tracking sites including GitHub and Google.

As an example, Issue 128796774 in Google’s issue tracker reported that stack traces in crash reports were unreadable in Android Vitals after the developer switched to deploying the Android app using App Bundles. Two external developers confirmed the issue, as did the Google engineer. Google claims to have fixed the problem; nonetheless the external developers say it was still occurring months later.

The analysis of this type of issue often includes cross-checking between Stack Overflow and Google’s Issue Tracker. Sometimes it also includes reviewing issues on GitHub. Where the issue has been seen in one or more of the app-centric case studies, potentially there is enough information from these sources to triage the issue for the respective project.

The grey material often provides additional examples and/or corroborates issues found in app-centric case studies.

Code and app-binary analysis

Source code provides a snapshot of raw ingredients used by the development team’s build process in order to generate an app. Analysis of the binary provides a snapshot of the final result of the build process of the released app.

Source code often includes at least one build ‘recipe’ so the build can also be evaluated and hopefully reproduced. The source code’s repository augments the source code by recording the historical evolution of the source code.

Code analysis in the context of mobile analytics involves searching for indications of the use of one or more in-app mobile analytics libraries in the project’s source code. The steps performed to identify indicators of the use of a mobile analytics library include:

1. Examine details in one or more gradle scripts where the analytics are added as a ‘dependency’; the version of the dependency provides useful meta-data on whether the analytics are actively maintained by the developers.
2. Examine initialisation of the analytics library in the source code; often this occurs in the Android app’s Application class.
3. Examine import statements in individual source code files that reference the analytics Java package(s).
4. Search for one or more wrapper class files. If these exist, then extend the search for these custom classes in step 3 and 5.
5. Search for calls to the original and any wrapper mobile analytics API classes.

After the five steps have been completed, the matching lines of source code are then available for analysis. The analysis can be performed for any snapshot of a codebase, i.e., for any commit made to the version control repository. Commands such as git blame provide information about the particular commit where lines of code were last updated and complement analysis of the snapshots.
4.3 Methodology

The application of five steps can be scripted. As an example, in joint research [9], a mix of manual and scripted steps was applied to enable the analysis of the use of Firebase mobile analytics in all the commits made to 57 opensource Android apps.

A useful confirmatory test to help establish the integrity of an app’s codebase is to build the app using the build scripts. There may be additional documentation of the build process available, e.g., in a README file incorporated into the code base.

Analysis of the app-binary was limited to free, public, online services, in particular the Exodus Privacy project and AppBrain. Both of these, for different reasons, analyse the binary file(s) for Android apps on Google Play.

The Exodus Privacy project focuses on privacy on behalf of end users and identifies the permissions requested by the app and the trackers embedded in the binary file [10]; AppBrain focuses on providing information to app developers [11]. In terms of this research, AppBrain provides statistics on the usage of various libraries across the population of apps in Google Play Store; Exodus Privacy provides details of the trackers used in the apps in the app-centric case studies.

4.3.6 Action research

Avison, Lau, Myers, and Nielsen explain the utility and importance of action research in order to establish the relevance of academic research by trying out theories with practitioners in real situations and in real organisations [181]. They recommend action research “because this particular qualitative research method is unique in the way it associates research and practice, so research informs practice and practice informs research synergistically.” [p.94]. Action research is particularly relevant for evaluation, when it “encourages researchers to experiment through intervention and reflect on their intervention and the implication of their theories.” [p.95].

This research applies the recommendation of Avison et. al. [181] where the research informs the practice of both app developers and the work of the developers of mobile analytics tools. Conversely, the research is enriched through understanding the practices and the potential for the application of mobile analytics to help improve the quality of the work of the app developers. The case studies include examples where the researcher’s mode of engagement was:

1. a consultant and/or an embedded developer: an active participant integrated into the project team;
2. a coach: of existing teams of app developers who applied the concepts;
3. an interviewer: of various development teams to learn of their practices and results;
4. an analyst/observer: performing static analysis of opensource code repositories [12].

Table 4.3 maps the roles to the app centric case studies. All the roles support communications with the project and allow the researcher to ask questions, offer suggestions including issues and/or code contributions,

[9]: Harty et al. (2021), ‘Logging Practices with Mobile Analytics: An Empirical Study on Firebase’

[10]: reports.exodus-privacy.eu.org/en/info/understand/

[181]: Avison et al. (1999), ‘Action research’

[12]: Analysis of proprietary code repositories is also possible, but not practicable for this research owing to confidentiality agreements.
and discuss findings with the developers for their case study. All include at least some observation and analysis.

The coach and embedded developer roles ended up being fairly similar. They both included organising a hackathon and a field experiment for the the Kiwix and Catrobat projects. The material differences were two-fold: a) the embedded developer role continued for several years, whereas the coach role was for several months, and b) the embedded developer role included contributions to the project’s code-base.

The consultant role included engineering leadership, mentoring developers and product owners in learning how they could integrate mobile analytics into their work practices, co-writing automated tests and related code, code reviews, and working with various mobile analytics tools throughout the assignment.

**Engagement** The action research was preceded by discussions with the project team (and its organisation) to determine:

- what research would be viable and productive;
- the role of the researcher;
- the depth, scope, range, and duration of the case study;
- concerns and constraints that needed to be addressed to protect all the stakeholders involved while also maintaining the integrity of the research;
- intellectual property rights, copyright, confidentiality, non-disclosure agreements, and so on.

The decisions were made mutually by the parties involved. Generally, the project’s team and its organisation set the limits and constraints. As Barroca et al. note [183, p.324], timeliness and relevance are vital to industry partners, while they also want to guard against the research being too intrusive or too demanding of their time or other resources. Therefore, the research needs to offer something of sufficient value, relevance, and timeliness to the project team and its organisation. For example, if they are aware their projects have excessively-high error rates, they may have the motivation to participate in the research in the hope of materially reducing the error rates, and furthermore they may seek an intervention in the guise of action research in order to achieve reductions in any excessively-high error rates. As the projects generally already have at least one form of mobile analytics, the incremental cost is low in terms of tooling.

**Wrap-up** The wrap-up of a case study included various actions such as safeguarding and archiving evidence; redacting personal details or anonymizing communications with the stakeholders; unsubscribing from services provided for a given case study; reviewing findings, analysis, and conclusions with the development teams (and with their organisation); preparing what is appropriate to publish in terms of evidence; and so on. This stage also provided an opportune period for retrospectives of the case study and for the research methods and outcomes, while the case study was still topical.
Observation and analysis

Observation is combined with analysis, both contemporaneous and post-hoc. Findings were presented to the project and development team, in order to: 1) provide value for the team members who were willing to contribute their time and who provided access to their analytics and other artefacts, and 2) encourage scrutiny and validation of the findings and observations.

In this research, observation included either or both development artefacts and analytics artefacts. One example is to observe the outputs from mobile analytics tools (the analytics artefacts) and what the developers do with these subsequently.

The analysis (as described earlier in the sense-making and sense-building sections) focused on ways the observations might usefully help the project in terms of improving the team's development practices and/or their artefacts.

**Contemporaneous analysis:** The nature of working with live mobile analytics data and real-world projects means there are likely to be important events that need to be found and processed rapidly, in order to preserve their value in terms of the case study. There are four key tasks: analysis, verification, course correction, and efficacious communications. All of these need to be performed on a continuous, iterative, and ongoing basis. Latency in the work must be kept to a minimum in order to increase the value of the results and the effects. The viability is governed by various factors, including: the working relationships with the project team set in the context of their working practices, research access to the various systems, and the characteristics of the event being reported via the mobile analytics tool(s).

**Post-hoc analysis:** Post-hoc analysis is more research-oriented; in contrast, analysis during the active case study is often more project-oriented. During the active case studies, there is a need to deal effectively with ephemeral events, data, actions, etc. on an opportunistic, often sporadic, basis; there is potentially a bias toward tactical findings and outcomes. From a research perspective, the active case study may appear messy, chaotic, and yet incomplete. The post-hoc stage provides the complementary opportunity for a more reflective, objective, and strategic perspective. It can help reduce inadvertent bias in the more immediate tactical work by seeking counterfactuals, alternatives, and/or mistakes and flaws.

By the post-hoc analysis stage, the active engagement with the project has tapered off, although in some cases additional updates may be available; e.g., some projects have provided ongoing access to mobile analytics and/or updates from the project team. Nonetheless, for the most part the evidence has been harvested, and any active interventions have ceased. The post-hoc analysis of collected records, actions, and results seeks to identify patterns in and across case studies, as well as to identify contradictory evidence, misconceptions, and potential omissions. Applying the six perspectives (illustrated in Figure 4.1) help to categorise and group various findings in the post-hoc analysis.
Conclusions, together with their supporting findings and their respective sources, were verified with the project team when feasible.

**Analysis of findings:** Conceptually, the findings that emerged from the analyses described above were themselves then analysed systematically in order to identify patterns or groupings (both within and across case studies): the findings were first analysed at a granular (low) level; these were then aggregated into higher level themes. Both the lower- and higher-level themes were counted to establish their relative frequency in the findings. To support the analysis, a spreadsheet was used that incorporates named ranges and formulae to reduce manual errors and make the analysis easier to scale and revise as any new findings emerged. Details on how the thematic analysis was performed are available in the Thematic Analysis appendix.

The approach I used was similar to the six-phase approach described in [184], albeit I used themes at both the lower and higher level analysis, along the lines of [185, pp. 280-281]. Visual modelling helped organise the themes and their relationships, as per [p. 280] in the same research. Originally focused on problem diagnosis and solution, their use here is generalised to express factors that characterise phenomena of interest. The visual models, in the form of Ishikawa (fishbone) diagrams were reviewed with several academic colleagues and refined iteratively together with suitable revisions to the various themes.

**Field experiment**

In this research, field experiments used real-world apps and their core code repositories. They were not as rigorous as controlled experiments, due to the nature of the development teams and their priorities. Nonetheless, they included a control app and an experiment app; experiments were performed on the experiment app, and mobile analytics results for both these apps were compared. They are also ecologically valid, [186, p.126], as they were situated in real-world challenges found in their core mobile apps.

The approach described by Ko et al. in [186] was not suitable for several reasons 16. My research didn't have the opportunity to compare two tools quantitatively, nor was it practical to perform quantitative research experiments, as none of the projects was interested in the complexity and demands of the type of controlled experiments illustrated in their research. Therefore, the field experiments were designed to suit the specific opportunities provided by the respective projects as follows:

- Hackathons were used to initiate the field experiments for the opensource projects. These were attractive to the development teams, partly as they were for a distinct period when the team was able to meet and enjoy the experience of collaborating.
- Both the opensource projects had multiple apps, including one that had the worst reliability of their apps (as measured by mobile analytics). The project leads saw value in helping arrange and support the hackathons. They also had apps which were suitable to act as the control for their hackathon.
Hackathon

Hackathons were initially conducted as a form of field experiment. However, they are described here in more detail, as the structure of the event and the follow-up activities were not restricted to a quasi-experiment, and the data was sufficiently rich to be treated differently. Hence, the hackathons are distinguished from the other field experiments in Table 4.2 above.

This research method applied the concepts of software development hackathons and added the use of mobile analytics tools as a source of information related to issues in the behaviour (i.e., failure data) of one of more of that team’s mobile apps.

In the terminology of Drouhard et al. [187, pp. 1-3] (as used in [188, p.3]), the hackathons were communal, as the participants were part of common development communities (as developers for the open-source apps to which they contributed); contributive, as there were common concerns about the issues the hackathons were intended to address, with a strong desire for impact; and catalytic, as one of the aims was to demonstrate a new approach to the use of a dataset (mobile analytics reports) and technology (mobile analytics tools).

No rewards were offered to participants beyond being at the respective hackathon and participating. Briscoe et al. [189, p.5] termed these ‘Single-Application’ hackathons. The participants were current members of the respective development team. The hackathons included the researcher and a highly-experienced leader and contributor in previous hackathons. They prepared much of the hackathons together, and in one case (PocketCode) co-led the hackathon. The app, topic, and focus were selected as part of the preparations. The work during the hackathon was determined by the participants, with guidance and suggestions from the organisers.

The outcomes of the hackathons included bug reports, and code changes, i.e., fixes intended to address some of the bugs found in the respective hackathon. Both the bug reports and the code changes were analysed over a period of several months to observe the effects of each hackathon. The mobile analytics reports were also monitored as new releases of the respective apps were launched to observe the effects of the code changes.

4.3.7 Summary of the analytics-centric methodology

The research needed to be grounded in real-world projects and with real-world app development teams; accordingly the methodology incorporates a mutually-beneficial, symbiotic, bi-directional connection where the iterative research is evaluated with and through these real-world projects and teams.

The analytics-centric methodology grounded in case studies corresponds in principle to the structure and premises of grounded theory [190] where patterns are induced, data is iterated to see if categories are appropriate, and identified patterns are compared to other data sets for checks and
balances. Although the research includes a variety of apps, development teams, and mobile analytics tools, the practical limitations of PhD research compared to the vast volumes of apps, mobile analytics tools, and so on means that the research and therefore the results cannot be definitive. The research will not reach saturation in terms of determining the transition point where additional cases would not contribute new insights.

4.4 Introducing the case studies

All the case studies involved application of sense-making, sense-building, and feedback mechanisms. There are two broad categories of case study: app-centric and tool-centric.

Each app-centric case study is centred around a single codebase; that codebase may be used in one app or in several, and the organisation may have additional code bases and associated apps. In the Catrobat case study, the additional app and codebase provide a useful contrast to the primary one in the case study.

The tool-centric case studies are centred on a single mobile analytics tool. Some of the tool-centric case studies emerged from the app-centric case studies; others emerged from mini-experiments or from grey materials.

4.4.1 Introducing the app-centric case studies

For the app-centric case studies in this research, every project included at least one actively-used Android app in Google Play\(^\text{17}\). The case studies range in the richness of their contributions to the research; Table 4.3 provides the research context for each of the seven app-centric case studies. The table has four groupings; these are for the four main research methods used in these case studies. The first grouping (and main research method) is for the interview-centric case studies, those without a planned intervention. The other three research methods all involve at least one planned intervention.

As Table 4.3 indicates, four project teams were interviewed to provide the developers’ perspectives on their use of various mobile analytics tools. Two of these (LocalHalo and GTAF) provided real-time access to at least one of these tools. GTAF also provided access to their bug tracking system which records issues they find using mobile analytics.

During this research I was already working as part of a development team for the Kiwix project. The other cases were accessed through personal recommendation; two were via academia (Catrobat and Greentech Apps Foundation) and the rest via software developers in industry.

Three project teams were subject to interventions. Of these, both Kiwix and Catrobat develop and work as opensource projects where access to their code, to their issue tracking, and to other aspects of their projects are public. They both also provided access to their mobile analytics tools. The last of these case studies, C1, is based on a mission-critical commercial product where a similar approach to applying mobile analytics was applied for the Android app element of the product.
### 4.4 Introducing the case studies

#### Table 4.3: App-centric cases: the research perspective

<table>
<thead>
<tr>
<th>Case Study</th>
<th>Main Research Role</th>
<th>Main Research Method</th>
<th>Research Opportunities</th>
<th>Research Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>GTAF</td>
<td>Interviewer</td>
<td>Ask the app devs</td>
<td>Additional perspective</td>
<td>Exploring the long tail</td>
</tr>
<tr>
<td>LocalHalo</td>
<td>Interviewer</td>
<td>Ask the app devs</td>
<td>Additional technologies</td>
<td>Hybrid Programming and tools</td>
</tr>
<tr>
<td>Moodspace</td>
<td>Interviewer</td>
<td>Ask the app devs</td>
<td>Small startup</td>
<td>Bootstrap view</td>
</tr>
<tr>
<td>Moonpig</td>
<td>Interviewer</td>
<td>Ask the app devs</td>
<td>Leading edge view</td>
<td>Mature, innovative, vanguard dev. practices</td>
</tr>
<tr>
<td>Smartnavi</td>
<td>Interviewer</td>
<td>Ask the app devs</td>
<td>Opensource codebase that incorporates Crash reporting and Firebase Analytics</td>
<td>Compare our analysis of their source code with the developer’s intentions</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interventions</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kiwix (Kiwix app)</td>
<td>Embedded</td>
<td>Field-experiment</td>
<td>The proof-of-concept</td>
<td>The treatment</td>
</tr>
<tr>
<td>Kiwix (WikiMed (EN))</td>
<td>Analyst/Observer</td>
<td></td>
<td>Control for the above app</td>
<td>The control</td>
</tr>
<tr>
<td>Kiwix (Custom apps)</td>
<td>Analyst/Observer</td>
<td></td>
<td>Evaluate scalability</td>
<td>pico generalisation</td>
</tr>
<tr>
<td>Catrobat (PocketCode)</td>
<td>Coach</td>
<td>Hackathon</td>
<td>Fabric Crashlytics</td>
<td>Compare Mobile Analytics with Clean Code</td>
</tr>
<tr>
<td>Catrobat (PocketPaint)</td>
<td>Analyst/Observer</td>
<td></td>
<td>Establish baseline</td>
<td>The control</td>
</tr>
<tr>
<td>CI</td>
<td>Consultant</td>
<td>Hybrid/Mixed</td>
<td>Large scale, complex, commercial</td>
<td>Mission-critical view</td>
</tr>
</tbody>
</table>

**Sources of ‘truth’:** In terms of the app-centric case studies there are several sources of ‘truth’ with respect to the use of mobile analytics. These include:

- The developers: including what they say they do, and how they used mobile analytics.
- The source code: including the build scripts and relevant configuration files. The source code + the build process generates one or more application binaries, and they may also generate and run various automated tests that exercise the app and/or the mobile analytics SDK(s).
- The app binary: which encapsulates and packages the software that is intended to run on end-user devices.

This research included interviewing some app developers and working with others to learn about their perceptions of using mobile analytics and the artefacts they created in the process.

Whenever the source code was available, it was studied to learn how mobile analytics had: a) been integrated, and b) been used to effect changes in the source code.

The Exodus Privacy project\(^1\) was used to detect whether the app binary included the appropriate mobile analytics SDKs.

### 4.4.2 Introducing the tool-centric case studies

The app-centric cases were complemented by empirical studies with several providers of mobile analytics tools where there was mutual interest in exploratory field studies; Table 4.4 provides the research context for the tool-centric case studies. They entries are grouped by the main research method.

The tool-centric studies range from those that were part of an app-centric case study (the first four listed in Table 4.4, i.e., Fabric Crashlytics, Microsoft App Centre, Sentry, and Google Play Console with Android...
Table 4.4: Tool-centric cases: the research perspective
IEEP = Informal Early Experience Program

<table>
<thead>
<tr>
<th>Case Study</th>
<th>Role of Researcher</th>
<th>Main Research Method</th>
<th>Research Opportunities</th>
<th>Research Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Firebase Analytics</td>
<td>Interviewer</td>
<td>Ask the app devs</td>
<td>Insights into maintaining a reliable app</td>
<td>Obtain expert user’s view of the most popular mobile analytics tool</td>
</tr>
<tr>
<td>Fabric Crashlytics</td>
<td>Analyst/Observer</td>
<td>Sense-making</td>
<td>Compare 2 Google Analytics tools</td>
<td>Triangulation</td>
</tr>
<tr>
<td>Microsoft App Centre Analyst/Observer</td>
<td>Sense-making</td>
<td>Crash &amp; Error analytics</td>
<td>Mobile cross-platform apps</td>
<td>Blue-chip alternative</td>
</tr>
<tr>
<td>Sentry</td>
<td>Analyst/Observer</td>
<td>Sense-making</td>
<td>Mobile analytics for React</td>
<td>Increase variety and coverage of tools</td>
</tr>
<tr>
<td>Google Play Console with</td>
<td>Analyst/Observer</td>
<td>Ask the tool devs</td>
<td>Mutual symbiotic cross-pollination</td>
<td>Learn about the providers’ perspectives</td>
</tr>
<tr>
<td>Android Vitals</td>
<td>Consultancy</td>
<td>Ask the tool devs</td>
<td>‘Behind the curtain’</td>
<td>Discover state of the art approach to improving the rigour of mobile analytics</td>
</tr>
<tr>
<td>Iteratively</td>
<td>Interviewer &amp; IEEP</td>
<td>Ask the tool devs</td>
<td>Explore state of the art novel tool</td>
<td>Insights into tools</td>
</tr>
<tr>
<td>Iteratively-&gt;Amplitude</td>
<td>Interviewer &amp; IEEP</td>
<td>Ask the tool devs</td>
<td>Explore state of the art novel tool</td>
<td>Insights into tools</td>
</tr>
</tbody>
</table>

Vitals). Of these, the Google Engineering team choose to engage further with the research based on the initial findings regarding behaviours and flaws in Android Vitals and Google Play Console.

The founders of Iteratively sought out the researcher, learned about the research, and were happy to share their tools and some of their commercial research. Shortly after Iteratively was acquired by Amplitude, the Iteratively SDK was incorporated into an updated version of one of the small local app experiments. In effect it was an informal, early-experience program (IEEP) which continued during the post-acquisition integration of Iteratively’s product into Amplitude and the migration of the mobile analytics account.

Finally, some minor contributions to two mobile analytics tools that include opensource elements which are not listed in this table (PostHog and Sentry) are presented in Section 5.9.2. Contributions to opensource mobile analytics projects, starting on page 119.

4.5 Ethical considerations

The research was informed by both the researcher’s professional experience as a software engineering practitioner, and professional codes of practice. During the research, I was a member of three relevant professional bodies: the British Computer Society (BCS), the IEEE, and the ACM, and I worked to abide by their respective codes of conduct [191, 192]. Where appropriate, ethics approval was obtained from the Open University’s Human Research Ethics Committee (HREC).

The ethical considerations implemented during this research can be described using the core concepts presented by Singer and Vinson, namely: “informed consent, scientific value, beneficence and confidentiality” [193, p.1178].

Informed consent: Consent was obtained from the development team leaders and development team members who participated; and consent was also obtained from their respective organisations, as appropriate. In
every case, they were explicitly aware of the research from the outset of discussions. Consent was obtained during the engagement discussions, and in some cases had to be extended during later phases, e.g., if the scope of the study increased and/or additional mobile analytics tools were introduced.

*De-facto* consent was also given in terms of the access provided to the tools and artefacts that teams and their organisations provided during the case study. They sometimes also placed constraints on aspects of the use of the materials obtained, i.e., consent was in some cases fine-grained and also context dependent.

Additionally, consent was obtained from the app development teams to share findings with the mobile analytics development teams and vice-versa as and where appropriate. This was not applicable in all the cases.

**Scientific value:** The importance and relevance of seeking ways to improve the quality of mobile apps and the processes used when developing those apps have been confirmed from multiple sources, including academia and industry. The importance of mobile analytics is evidenced by the practices of app developers who include mobile analytics in over 75% of all Android apps in Google Play, and by the ongoing development of a multitude of mobile analytics tools and related services. This research aims to contribute to knowledge about the practices, the tools, and the outcomes of using mobile analytics tools as part of development practices.

**Beneficence:** Beneficence aims to maximise the overall benefits for all the stakeholders and harm none. This includes the people who participate in the development teams, their organisations (where applicable), and the end users of their mobile app(s). The different beneficiaries of the research can be summarised as follows: (1) app developers, by helping them make effective use of mobile analytics; (2) organisations involved, by helping improve the quality of their apps; (3) analytics tool developers, by providing insights into tool use and potential improvements; and (4) end-users, by potentially improving the app’s performance and reliability downstream.

**Confidentiality:** The confidentiality of the participants and also of the information provided and/or gleaned during the case study was protected unless a) the work is in the public domain, or b) permission was granted to make the information public, e.g., as part of this research. As the apps run on end-users’ mobile devices, the risks of the data collection and the use of that data were also considered throughout the research. In particular, although some of the apps of the projects do contain PII, no PII was collected or analysed in the mobile analytics from a research perspective.

**Agency of participants and their organisations:** ‘Agency’ is the concept that the organisations and the relevant people were free to choose whether they wished to participate in the research, and if so potentially the nature of their involvement and/or the nature of the research.
Some candidate projects declined to participate in the research on behalf of their project or organisation for various reasons. A common reason was lack of time on their part; another was that some candidate projects perceived that the research would not be acceptable to their organisation, for instance owing to confidentiality or business risk. The participating projects chose their model of engagement, which meant the researcher needed to negotiate the modes of engagement to balance ways of working necessitated by the research with the industry practices in the domain of mobile app development.

4.6 Validity and rigour

This section discusses the validity and rigour of the research in terms of the methods adopted for data collection and analysis.

The use of case studies to explore how analytics affect mobile app development processes and the apps themselves results in a natural trade-off between the internal and external (including ecological) validity of the research. Because the research is being conducted in the context of mobile application development projects, with limited ability to control for external factors, the internal validity of the overall findings is low. However, the aim of the research is not to prove hypotheses about the causal relationships between mobile analytics and the quality of the software developed. Instead it is to explore systematically the phenomena relating to the effect of mobile analytics on development processes and artefacts, and use these findings to support the formulation of hypotheses for further study. In order to ensure that such exploration is grounded in real-world experiences, it was considered appropriate to give priority to external (and ecological) validity over internal validity.

The approach adopted to maximise the rigour of the research is to ensure that the methods chosen for data collection and analysis are repeatable. While the specific case studies covered in the research were selected opportunistically, the methods used to collect data and analyse it can be carried out by other researchers. The remainder of this section provides some further details of how the research design supports external validity and repeatability.

**External validity:** was high for both the app and the tool case studies. While it was not practical or viable to control all the factors, the use of control apps, and tracking the changes made to the source code, provided probable causation in terms of increases in the stability of the apps being subject to the interventions.

At least some of the results from the micro-experiments also led to subsequent outcomes in the real-world apps and tools. These outcomes help corroborate the external validity of these more controlled experiments.

At least two of the software utilities that were developed as part of this research have been used by other projects; Vitals Scraper was used at Moonpig, and AndroidCrashDummy was enhanced and used by a corporation (details of whom are covered by private communications).

It is also noteworthy that all the case studies are real-world projects with real-world engineering desires to apply mobile analytics, i.e., “illustrating
a tool’s benefits (or lack thereof) on a real software engineering activity taken from practice” [186, p.126]. Further, the case studies represent a variety of mobile application development contexts. Considered together, these support the ecological validity of the research.

**Repeatability:** As mentioned here and elsewhere in the dissertation, the software developed as part of this research has been released under a permissive opensource license to encourage and facilitate further research. A set of the artefacts generated by Vitals Scraper has been preserved and made available to the research community.

### 4.7 Chapter summary

The methodology has been chosen to maximise the viability of answering the primary research question in real-world projects and contexts where access to the tools is granted by project teams and their respective organisations, and where the development teams actively engage and support the case studies. It challenges the researcher’s understanding by comparison with other data and sources and through vigilant attention to potentially contradictory evidence.

The methodology also accommodates complementary research that augments the case studies, for instance with the analysis of various opensource Android codebases) introduced in Section 5.10. Augmenting the app-centric case studies: sourcecode analysis.

The six perspectives inform the methodology by helping ensure that combinations of the various methods address the three key objects of analysis, and also address the temporal dimension of determining what is and what could be.

The next chapter introduces each of the case studies.
Overview of the case studies

Scientific research is one of the most exciting and rewarding of occupations.

Frederick Sanger - 1918-2013

This chapter introduces each of the app-centric and tool-centric case studies using a consistent structure to make them easy to comprehend and to facilitate comparison. For each of the app-centric case studies (except the commercial case study C1) it uses two summary tables to do so. The first table presents the key-facts pertaining to that case study, the second presents the data sources. These are augmented with brief descriptions to provide additional context. Each of the overviews for the tool-centric case studies includes a table with the key-facts augmented with a contextual summary together with the data collected and methods used for collecting material for this mobile analytics tool.

The depth of each section varies to focus on material that’s of consequence to the thesis. The findings for the case studies and the additional research are presented in the subsequent three chapters. The Index provides a mapping of where each case study contributes to the thesis.

The case studies are in four groups that present complementary research that helps to fill various gaps between the app-centric and tool-centric case studies. The groups are:

1. App-centric case studies that do not have interventions (Sections 5.1, to 5.5).
2. App-centric case studies that do have interventions (Sections 5.6 to 5.8).
3. Various field experiments (Section 5.9) and source code analysis (Section 5.10).
4. Tool-centric case studies (Sections 5.11 to 5.16).

Finally, section 5.17 briefly introduces six more mobile analytics tools and another Android app as these provide miscellaneous minor contributions in the subsequent three chapters.

The app-centric case studies are presented in the same order as Table 4.3 for the app-centric cases and then Table 4.4 for the tool-centric cases.

The nature of the case studies means they include observations and findings that occurred at the time. There may be situations where there are differences between these observations and findings and those of other users of these tools.
5.1 App-centric: GTAF

A minor case study that contributed several findings including their fragmented use of mobile analytics, their public issues database with bugs found by Firebase Analytics, and the team’s public acknowledgement that they were able to improve the reliability of their most active mobile apps through using mobile analytics.

Case study participants: 2 people: the project’s co-founder and lead developer; and additional background discussions with another developer who was another post-graduate student during the case study.

Interview format: questions were prepared for the interview with the lead developer which led to an open discussion. This interview used video conferencing (Google Meet) and lasted 50 minutes. This was combined with email correspondence both before and after the interview.

Data collection and use: the data sources obtained in this case study are summarised in Table 5.2, they include:

- **Contemporaneous notes:** that led to various of the presented findings. These were shared and validated via the Google document I shared with the project’s founders (one was the person I interviewed), others were validated during the email correspondence. Note: the main interviewee was unwell and I did not receive validation of every finding.
- **Emails:** Corroborated findings.
- **Mobile analytics:** Interactive screenshots and Vitals-scraper data from Google Play Console and Android Vitals for reports, failures, statistics,
- **Issues database:** Analysed to find issues that included references to mobile analytics as a source of that issue.

Greentech Apps Foundation (GTAF) is a UK-based charity that provides Islamic apps free of charge and without in-app advertising. The project started in 2016 with the aim of enabling people to learn the Quran in the local language – Bangla – in Bangladesh. The project was started by a self-taught Android developer and his cousin Yemin, at the time an undergraduate student in computer science, who is now employed by the project in a hybrid role of software developer and project manager. Table 5.1 summarises the key facts for this case study.

The project team hosts their development artefacts on gitlab.com, and they maintain their issues in a publicly-available online database at https://gitlab.com/greentech/; the source code is private. There is a mix of paid developers (through donations to the charity) and volunteers (often part-time). The developers of some of the less-active apps appear relatively autonomous, which includes their choice and any use of mobile analytics.

Three codebases were in ongoing active development and used to generate four Android apps (including Dua and Zikr in English and Bangla) (Al Quran, Hadith Collection, and Dua & Zikr, which is also released separately in Bangla Dua and Zikr (Hisnul Muslim) in Bengali); and they

*The project have subsequently released several of their apps on other platforms, see https://gtaf.org/apps.*
Table 5.1: Case Study key facts: GTAF

<table>
<thead>
<tr>
<th>Website</th>
<th><a href="https://gtaf.org/">https://gtaf.org/</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>Founded</td>
<td>2016</td>
</tr>
<tr>
<td>Business Domain</td>
<td>Not-for-profit.</td>
</tr>
<tr>
<td>Business type</td>
<td>Educational foundation.</td>
</tr>
<tr>
<td>Technologies</td>
<td>Android apps*</td>
</tr>
<tr>
<td>React Native</td>
<td></td>
</tr>
<tr>
<td>Source code</td>
<td>Closed and not available for research</td>
</tr>
<tr>
<td>Analytics used by team</td>
<td>Firebase, OneSignal, Google Crashlytics, Google Play Console</td>
</tr>
<tr>
<td>Development Practices</td>
<td>Small hybrid development team</td>
</tr>
<tr>
<td>User base</td>
<td>1,000,000’+ for their 10 Android apps</td>
</tr>
<tr>
<td>Installations</td>
<td>1,000,000’s for their 10 Android apps</td>
</tr>
<tr>
<td>Source of case study</td>
<td>Via a fellow PhD researcher who was a software developer on the project.</td>
</tr>
<tr>
<td>Catalyst for the case study</td>
<td>Excessively high crash rates for several of their apps piqued their interest in the research.</td>
</tr>
<tr>
<td>Research methods</td>
<td>Online interview and email discussions, etc.</td>
</tr>
<tr>
<td>Participants</td>
<td>2 people actively participated, the other co-founder also received a copy of my findings at his request</td>
</tr>
<tr>
<td>Analytics collected</td>
<td>Google Play Console with Android Vitals</td>
</tr>
<tr>
<td>Research software</td>
<td>None applicable?</td>
</tr>
<tr>
<td>Additional data collected</td>
<td>Direct access to Google Play Console with Android Vitals, and to the public, issue database. Interview notes and emails.</td>
</tr>
<tr>
<td>Active period</td>
<td>June 2020 to September 2020</td>
</tr>
<tr>
<td>Post-hoc analysis</td>
<td>Included ongoing access to their Google Play Console for 18 months.</td>
</tr>
</tbody>
</table>

planned to revamp two more of the apps (Islamic Quiz and Meaningful prayers (salat) in Bengali.

The team occasionally used Firebase TestLab\(^1\) to test some of the apps, and autonomous ‘Robo testing’\(^2\), performed automatically by the test lab, triggered various crashes in the apps being tested. One such example was where an app was missing a ‘resource’. The team fixed the build by adding the missing resource but did not explicitly retest the app afterward in Firebase.

**Corroboration:** A Google Document was written and shared with Project leads of the Greentech Apps Foundation (GTAF) project.

\(^1\): firebase.google.com/docs/test-lab

\(^2\): firebase.google.com/docs/test-lab/android/robo-ux-test
Table 5.2: GTAF: data sources

<table>
<thead>
<tr>
<th>Data Source</th>
<th>Records</th>
<th>Volumes</th>
<th>Analysis method</th>
<th>Contribution</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-study interview, with core developer</td>
<td>Contemporaneous notes</td>
<td>1</td>
<td>Ask the app devs</td>
<td>Set scope &amp; direction</td>
<td>Online call</td>
</tr>
<tr>
<td>Analytics tools &amp; artefacts</td>
<td>Interactive screenshots &amp; Vitals-scraper outputs</td>
<td>$10^1$</td>
<td>Beacon finding, drill down, across case comparisons, observation &amp; analysis.</td>
<td>Indications of the development team’s attention to the crash rate, insights into the performance of their apps, corroboration of findings across various case studies.</td>
<td>Google Play Console with Android Vitals.</td>
</tr>
<tr>
<td>Mid-study communications with developers</td>
<td>GMail</td>
<td>$10^1$</td>
<td>Ask the app devs</td>
<td>Feedback, and sense-making.</td>
<td>Email conversations corroborated findings.</td>
</tr>
<tr>
<td>Development artefacts</td>
<td>Issues database</td>
<td>$10^2$</td>
<td>Observation and Analysis, analysis of development artefacts.</td>
<td>Corroboration of what the development team say they do in terms of using mobile analytics.</td>
<td>Public GitLab repo.</td>
</tr>
</tbody>
</table>

5.2 App-centric: LocalHalo

This mid-sized case study provided longitudinal access to one of the commercial yet opensource mobile analytics offerings: Sentry. It also included a cross-platform mobile app and surfaced how app crashes were generally automatically recovered by the cross-platform runtime and invisible to platform-level mobile analytics.

**Interview format:** The interviews started with an open introductory discussion with CEO on a videoconferencing call that lasted 25 minutes. The next discussion was in-person and lasted approximately 60 minutes. This went into more detail about the research and in mobile analytics. In a subsequent 60 minute videoconferencing call with the CTO three primary topics were raised and discussed: 1) their use of mobile analytics generally, 2) failure reporting and management, and 3) their development team practices.

**Data Collection and use** included:

- **Contemporaneous notes:** These recorded the use of three distinct mobile analytics services: Mixpanel for marketing and business development, Sentry for technology-facing issues, and Google Play Console with Android Vitals. Key points were shared by email to check understanding and for accuracy.

- **Correspondence emails:** during the case study primarily discussed errors that occurred during the case study. These corroborated findings from mobile analytics and automated emails.

- **Automated emails:** generated by Sentry on an ongoing basis included weekly summary reports for the website and for their cross-platform mobile app. A subset of these were compared with the results from Android Vitals, particularly when Android reported crashes in the Local Halo app and conversely Sentry had no data for the app on those dates.

- **Mobile analytics:** Interactive screenshots and Vitals-scraper data from Google Play Console and Android Vitals for reports, failures, statistics, which were available for six months. These were collected both using Vitals Scraper and also reviewed interactively online during the case study. Access to the company’s Sentry Mobile Analytics was provided indefinitely and only terminated when
### Table 5.3: Case Study key facts: Local Halo

<table>
<thead>
<tr>
<th>Website</th>
<th><a href="https://www.localhalo.com/">https://www.localhalo.com/</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>Founded</td>
<td>2018</td>
</tr>
<tr>
<td>Business Domain</td>
<td>Digital neighbourhood groups in UK.</td>
</tr>
<tr>
<td>Business type</td>
<td>Startup, two co-founders: CEO and CTO</td>
</tr>
<tr>
<td>Technologies</td>
<td>React Native for cross-platform Android and iOS development</td>
</tr>
<tr>
<td>Source code</td>
<td>Closed and not available for research</td>
</tr>
<tr>
<td>Analytics used by team</td>
<td>Sentry, Mixpanel, Google Play Console</td>
</tr>
<tr>
<td>Development Practices</td>
<td>Not explicit, a small distributed team</td>
</tr>
<tr>
<td>User base</td>
<td>7,000 registered users and between 1k to 2k monthly active users in Jan 2020.</td>
</tr>
<tr>
<td>Installations</td>
<td>1,000’s for the Android app</td>
</tr>
<tr>
<td>Source of case study</td>
<td>One of their team knew of my professional work and introduced me to the CEO.</td>
</tr>
<tr>
<td>Catalyst for the case study</td>
<td>The CEO was keen to support the research.</td>
</tr>
<tr>
<td>Approvals</td>
<td>Internally approved within the core development team.</td>
</tr>
<tr>
<td>Research methods</td>
<td>Interview, email discussions, bug analysis, use of mobile analytics</td>
</tr>
<tr>
<td>Participants</td>
<td>2 people, the CEO and the CTO</td>
</tr>
<tr>
<td>Analytics collected</td>
<td>Live access to: Sentry, Google Play Console with Android Vitals</td>
</tr>
<tr>
<td>Research software</td>
<td>Vitals-Scraper used to preserve results</td>
</tr>
<tr>
<td>Additional data collected</td>
<td>Interview notes, emails with the CTO, and automated emails from their mobile analytics services</td>
</tr>
<tr>
<td>Active period</td>
<td>Jan 2020 to June 2020</td>
</tr>
<tr>
<td>Post-hoc analysis</td>
<td>Included ongoing access to Sentry until late 2021.</td>
</tr>
</tbody>
</table>

Sentry changed their free-of-charge service offering and removed support for multiple logins.

> **Issues database:** Not available, by agreement with the CTO.

Google Play Console and Android Vitals reports, Sentry reports. Access was initially provided for three months, this was extended for another three months. The Sentry service continued to be available for over a year as no-one actively disabled it

LocalHalo was a startup based in London who made a social network for neighbours with developers in Ukraine, London, and Kazakhstan. [194]. Table 5.3 provides an overview of this case study.

The development team used the Expo development platform to create native apps that ran on Android and iOS apps. The mobile app was written in React Native (the associated website is likely to have been written using React js and also instrumented using Sentry Analytics which were also available for research purposes). These apps were released on Google Play and Apple’s App Store respectively. The CTO was actively involved in writing and maintaining the source code and was supported by developers in three locations, in the Ukraine, London, and Kazakhstan. [194]. Data in the Sentry mobile analytics reports indicate there was at least one distinct developer in addition to the CTO.

Little additional information was available during conversations in terms of their development or release practices for their mobile apps. One observation is Expo claims to automate the release process to the app stores so the LocalHalo development team may have relied and used the Expo service. And Sentry provided reports on the release numbers and their rollout.

**Data sources:** Table 5.4 provides details of the data sources obtained during this case study.
Table 5.4: LocalHalo: data sources

<table>
<thead>
<tr>
<th>Data Source</th>
<th>Records</th>
<th>Volumes</th>
<th>Analysis method</th>
<th>Contribution</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-study interviews</td>
<td>Contemporaneous notes†</td>
<td>3</td>
<td>Ask the app devs.</td>
<td>Set scope and direction</td>
<td>1-to-1 meetings with founders: 1 was in-person, 2 were online. Initiated by the researcher.</td>
</tr>
<tr>
<td>Mid-study communications with developers</td>
<td>GMail</td>
<td>$10^1$</td>
<td>Ask the app devs.</td>
<td>Insight into the Expo bug</td>
<td>Google Play Console with Android Vitals.</td>
</tr>
<tr>
<td>Analytics tools &amp; artefacts</td>
<td>Interactive screenshots &amp; Vitals-scraper outputs</td>
<td>$10^1$</td>
<td>Beacon finding, drill down, across case comparisons, observation &amp; analysis.</td>
<td>Insights into the reporting effects of the Expo bug and the reporting provided for React-Native apps</td>
<td></td>
</tr>
<tr>
<td>Analytics tools &amp; artefacts</td>
<td>Interactive screenshots of the Sentry GUI</td>
<td>$10^1$</td>
<td>Beacon finding, drill down, across tool comparisons†, observation &amp; analysis.</td>
<td>Insights into Sentry’s reporting</td>
<td>Access continued until Sentry removed multi-account access from their free tier.</td>
</tr>
<tr>
<td>Analytics tools &amp; artefacts</td>
<td>Sentry automated emails in GMail</td>
<td>$10^2$</td>
<td>Beacon finding, drill down, observation and analysis.</td>
<td>Insights into Sentry’s reporting, dev practices, &amp; cross-platform reporting</td>
<td>ditto. NB: they continue to send weekly reports by email.</td>
</tr>
</tbody>
</table>

Corroboration: was provided by email to the CTO and informal discussions were maintained with the CEO for a period (until the Coronavirus lockdowns in the UK intensified).

5.3 App-centric: Moodspace

This minor case study provided insights into how one of the platform-level analytics could be made more useful to the development team. It also illustrated how developers can sometimes design their mobile app to be both highly performant and reliable in use.

Case study participants the CTO of this startup was the sole participant from the company.

Interview format: started with structured questions written up in emails on their use of Android Vitals in particular; these were answered by email together with snapshots of their analytics reports. The interview was conducted by email with the bulk of the information provided on 13th and 15th June 2019.

Data collection and use: All the data was collected by email, Snapshots of Android Vitals and Google Play Console, answers provided via email. These are summarised in Table 5.6.

The data analysed consisted of:

- **Emails**: Their use of Android Vitals and their assessment of possible improvements to Google Play Console and Android Vitals.
- **Mobile analytics**: Interactive screenshots of Google Play Console and Android Vitals.

These were augmented by searches of the literature, grey data, and grey literature.
Moodspace is an Android app aimed at improving mental health through various exercises incorporated into the app. It was released in 2019, with over 150K downloads by early 2020 [195]. The interviewee, was ‘the software developer, co-founder, and runner of the company’ [195] so he combined technical and operational responsibilities in his Chief Technology Officer (CTO) role. He is an experienced app developer and also trained as a chemical engineer. The CTO was the sole developer.

The source code, issues database, and mobile analytics were all private.

At the time of the interview the team had developed a complete replacement for the Android app which was due to be launched in a couple of months (around August 2019). The startup was subsequently able to raise a round of funding and grow to six people. The project later returned to be a side-project which is maintained and updated [196].

The app had a strong positive User Experience rating of 4.18/5.00 which corroborates their objective to provide a beautiful and peaceful mobile app.

**Corroboration**: of the analysis was offered to the CTO, he replied to say he was too busy to follow up.

---

**Table 5.5: Case Study key facts: Moodspace**

<table>
<thead>
<tr>
<th>Website</th>
<th><a href="https://moodspace.org/">https://moodspace.org/</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>Founded</td>
<td>2019</td>
</tr>
<tr>
<td>Business Domain</td>
<td>Health</td>
</tr>
<tr>
<td>Business type</td>
<td>Online, in-app purchases.</td>
</tr>
<tr>
<td>Technologies</td>
<td>Android</td>
</tr>
<tr>
<td>Source code</td>
<td>Closed and not available for research</td>
</tr>
<tr>
<td>Analytics used by team</td>
<td>Firebase Analytics, Firebase Crashlytics, Google Play Console</td>
</tr>
<tr>
<td>Development Practices</td>
<td>Sole software developer</td>
</tr>
<tr>
<td>User base</td>
<td>10,000’s for the Android app</td>
</tr>
<tr>
<td>Installations</td>
<td>100,000’s for the Android app</td>
</tr>
<tr>
<td>Source of case study</td>
<td>One of their team knew of my professional work and introduced me to the CTO</td>
</tr>
<tr>
<td>Catalyst for the case study</td>
<td>The CTO was happy to support the research.</td>
</tr>
<tr>
<td>Approvals</td>
<td>Directly from the interviewee, the CTO</td>
</tr>
<tr>
<td>Research methods</td>
<td>Email interview &amp; follow-up discussions</td>
</tr>
<tr>
<td>Participants</td>
<td>The CTO who was also the main developer of the Android app</td>
</tr>
<tr>
<td>Analytics collected</td>
<td>Google Play Console with Android Vitals</td>
</tr>
<tr>
<td>Research software</td>
<td>None applicable</td>
</tr>
<tr>
<td>Additional data collected</td>
<td>Exodus Privacy project reports.</td>
</tr>
<tr>
<td>Active period</td>
<td>June 2019</td>
</tr>
<tr>
<td>Post-hoc analysis</td>
<td>Material provided during the active period was combined with additional Grey Material, particularly for objectbox, an in-memory ORM database used in the app</td>
</tr>
</tbody>
</table>

Interestingly standard Google Search finds peer reviewed papers that contain the name of the app, whereas Google Scholar struggles to find them. TBA how to classify Exodus Privacy and similar services that analyse app binaries. For the moment, development artefacts seems to be the best match of the existing data sources. [reports.exodus-privacy.eu.org/en/reports/search/boundless.moodgym](https://reports.exodus-privacy.eu.org/en/reports/search/boundless.moodgym)
Table 5.6: Moodspace: data sources

<table>
<thead>
<tr>
<th>Data Source</th>
<th>Records</th>
<th>Volumes</th>
<th>Analysis method</th>
<th>Contribution</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-study interview &amp; Mid-study communications with developers</td>
<td>GMail</td>
<td>10</td>
<td>Ask the app devs.</td>
<td>Effectively the interview, albeit using emails.</td>
<td>We ended up simply using emails rather than arranging a synchronous call and then continued the discussion using email.</td>
</tr>
<tr>
<td>Analytics tools &amp; Grey Data</td>
<td>Interactive screenshots</td>
<td>6</td>
<td>Ask the app devs.</td>
<td>Evidence, and allows for comparisons.</td>
<td>Google Play Console with Android Vitals. The app was made fully free in response to the Covid-19 pandemic.</td>
</tr>
<tr>
<td>Grey Data</td>
<td>Tweets</td>
<td>3</td>
<td>Grey Data</td>
<td>Triangulation</td>
<td>The app has been studied in various peer-reviewed papers. (9) There are various discussions about the efficacy and suitability of this and similar apps. Their 4 snapshots indicate a variety of mobile analytics have been incorporated.</td>
</tr>
<tr>
<td>Academic Literature</td>
<td>Peer reviewed publications</td>
<td>3</td>
<td>Secondary research.</td>
<td>Additional context</td>
<td></td>
</tr>
<tr>
<td>Grey Literature</td>
<td>Online articles</td>
<td>10</td>
<td>Secondary research</td>
<td>Additional context</td>
<td></td>
</tr>
<tr>
<td>Development artefacts</td>
<td>Exodus Privacy project online reports</td>
<td>4</td>
<td>Grey Data</td>
<td>Details of mobile analytics integrated into the app</td>
<td></td>
</tr>
</tbody>
</table>

5.4 App-centric: Moonpig

This case study contributed a long-term example of a successful and growing e-commerce business who relied extensively on their core mobile app to deliver much of their revenue. Their use of Firebase Analytics and Android Vitals provided a strong example of how a software engineering team could actively manage and constrain any failures in their Android app on an ongoing basis.

**Case study participants:** primarily one of their lead developers; this was supplemented by informal discussions with several of their development team at events hosted by Moonpig. Their legal and marking team also agreed the findings of the research could be published.

Moonpig is an e-commerce business in Europe that sells greeting cards and related gifts online; they operate in the United Kingdom, USA and Australia. They have a highly rated mobile app, with overall ratings of 4.8/5 in Google Play and the Apple App Store. Note some portions of this case study were published in [3].

The software development department demonstrated high performance engineering in how they approached both public contributions, for instance through hosting Coding DoJos, and in terms of their use of mobile analytics to quickly triage and address pertinent issues.

The engineering organisation consisted of various teams, including a team for the Android app. At the time of the case study, the Android app combined several generations of their architecture and used various third-party libraries. One of these third-party libraries, Robospice, led to a higher crash rate for their Android app on newer releases of Android. The details are discussed in Section 6.2.1. Engineering tradeoffs.

Their software engineering team have been actively involved in encouraging the wider software engineering community to learn and practice good software development practices, for example by hosting Coding DoJos. (9) They practiced similar software development practices when de-
Table 5.7: Case Study key facts: Moonpig

<table>
<thead>
<tr>
<th>Website</th>
<th><a href="https://www.moonpig.com/uk/">https://www.moonpig.com/uk/</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>Founded</td>
<td>2000</td>
</tr>
<tr>
<td>Business Domain</td>
<td>Greeting cards and gifts</td>
</tr>
<tr>
<td>Business type</td>
<td>e-commerce</td>
</tr>
<tr>
<td>Technologies</td>
<td>Native apps, Robospice, AWS, GraphQL, nodeJS, Commercetools, ContentStack, ...</td>
</tr>
<tr>
<td>Source code</td>
<td>Closed and not available for research</td>
</tr>
<tr>
<td>Analytics used by team</td>
<td>Firebase, Google Play Console</td>
</tr>
<tr>
<td>Development Practices</td>
<td>High performance engineering, ATDD, micro-services architecture</td>
</tr>
<tr>
<td>User base</td>
<td>100,000’s for the Android app</td>
</tr>
<tr>
<td>Installations</td>
<td>1,000,000+ for the Android app</td>
</tr>
<tr>
<td>Source of case study</td>
<td>Meeting one of the development leads at meetups and peer workshops. Their genuine interest in the effects of the hypothesis. Both the head of engineering and communication manager approved the case study and gave permission for the material to be used.</td>
</tr>
<tr>
<td>Catalyst for the case study Approvals</td>
<td></td>
</tr>
<tr>
<td>Approvals</td>
<td></td>
</tr>
<tr>
<td>Research methods</td>
<td>In person interviews, email discussions, remote testing</td>
</tr>
<tr>
<td>Analytics collected</td>
<td>Google Play Console with Android Vitals</td>
</tr>
<tr>
<td>Participants</td>
<td>Primarily the lead developer of the Android app several developers contributed informally</td>
</tr>
<tr>
<td>Research software</td>
<td>They used Vitals-Scraper, otherwise none applicable.</td>
</tr>
<tr>
<td>Additional data collected</td>
<td>Interview notes and emails</td>
</tr>
<tr>
<td>Active period</td>
<td>June 2019 to July 2019</td>
</tr>
</tbody>
</table>

Developing their production software, for instance in applying Acceptance Test Driven Development (ATDD) [197].

**Interview format**: initial interviews started as open-ended discussions about software quality practices at Moonpig, then extended into their use of mobile analytics and their app development practices.

**Interview conducted**: at least six in-person meetings supplemented by video calls and email discussions during the case study.

**Data collection and use**: analytics reports including snapshots of Android Vitals and Google Play Console were provided indirectly by email. Emails were also used for *ad-hoc* questions, discussions, and answers. These were augmented by in-person and online meetings before, during, and after the active period of the case study.

**Data analysed** included:

- **Contemporaneous notes**: Provided concrete examples of how they were using mobile analytics, of failures, and how they addressed those failures.
- **Emails**: Ongoing updates on their use of mobile analytics, bug reporting, a discussion of a crash reported in a review by an end-user of their Android app, and the company’s decision to halt their support of the research around the time the company listed on the stock market.
- **Mobile analytics**: Interactive screenshots and Vitals-scraper outputs provided snapshots of Google Play Console, Android Vitals, and Firebase Analytics.
- **Issues database**: Not available as their issue database was proprietary and confidential.
### Overview of the case studies

**Table 5.8: Moonpig: data sources**

<table>
<thead>
<tr>
<th>Data Source</th>
<th>Records</th>
<th>Volumes</th>
<th>Analysis method</th>
<th>Contribution</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-study interviews, mid-study communications with developers, &amp; walkthroughs</td>
<td>Contemporaneous notes</td>
<td>$10^3$</td>
<td>Ask the app devs</td>
<td>Multiple insights</td>
<td>A mix of in-person meetings and video calls</td>
</tr>
<tr>
<td>Mid-study communications with developers</td>
<td>GMail</td>
<td>$10^2$</td>
<td>Ask the app devs</td>
<td>Cross-checking understanding, additional insights</td>
<td>Email conversations that helped support published research</td>
</tr>
<tr>
<td>Analytics tools &amp; artefacts</td>
<td>Interactive screenshots from Google Play Console with Android Vitals &amp; Vitals-scaper outputs</td>
<td>$10^3$</td>
<td>Sensemaking, ask the app devs.</td>
<td>External verification of vials-scaper</td>
<td>They ran vitals-scaper to evaluate whether it worked for other people</td>
</tr>
<tr>
<td>Analytics tools &amp; artefacts</td>
<td>Interactive screenshots from Firebase Analytics</td>
<td>3</td>
<td>Ask the app devs</td>
<td>Comparison of crash reporting in two mobile analytics tools.</td>
<td>Screenshots from Firebase Analytics and Android Vitals provided an opportunity to compare their outputs</td>
</tr>
</tbody>
</table>

**Data sources:** a summary of the data sources obtained in this case study are provided in Table 5.8.

**Corroboration:** their lead developer reviewed peer-reviewed research published during the case study and was granted access to the PhD thesis at his request. The findings were corrobated on an ongoing basis between 2019 and 2023.

### 5.5 App-centric: SmartNavi

This minor case study provided insights into the use of Firebase Analytics for an app that ran as a background service on Android. Through the opensource codebase it provided additional, referenceable, history of the changes to the project’s code for crash and in-app analytics.

**Case study participants:** there was only one participant from the project, the main developer.

SmartNavi is an unusual opensource project that replaces GPS for navigation when the users are walking. It uses significantly less power than using a true GPS provider. Google promoted the project as an Android experiment [https://experiments.withgoogle.com/smartnavi](https://experiments.withgoogle.com/smartnavi).

**Interview format:** the contents of the interview questions were informed by prior analysis of the app’s source code on GitHub.com as part of collaborative research on the use of Firebase Analytics for logging purposes. It was designed as a single call that would cover as much of the ground as practical rather than needing to rely on multiple calls (which might be rejected by the interviewee as too burdensome).

**Interview conducted:** 60 minutes on a videoconference call (Google Meet). It was an open-ended discussion that started with the history and rationale for his app; his use of Firebase Analytics in the app, crash reporting, challenges of developing code that needed to run as a background service on Android - where newer releases of Android had become increasingly restrictive of background processes in general.
Table 5.9: Case Study key facts: SmartNavi

<table>
<thead>
<tr>
<th>Website</th>
<th><a href="https://smartnavi.app/home">https://smartnavi.app/home</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>Founded</td>
<td>2014</td>
</tr>
<tr>
<td>Business Domain</td>
<td>Maps &amp; Navigation</td>
</tr>
<tr>
<td>Business type</td>
<td>None, a student project that grew</td>
</tr>
<tr>
<td>Technologies</td>
<td>Android</td>
</tr>
<tr>
<td>Source code</td>
<td>Opensource <a href="https://github.com/Phantast/smartnavi">https://github.com/Phantast/smartnavi</a></td>
</tr>
<tr>
<td>Analytics used by team</td>
<td>Fabric Crashlytics, Firebase Analytics, Google Play Console</td>
</tr>
<tr>
<td>Development Practices</td>
<td>Main developer who accepts pull requests</td>
</tr>
<tr>
<td>User base</td>
<td>10,000’s for the Android app</td>
</tr>
<tr>
<td>Installations</td>
<td>50,000+ for the Android app</td>
</tr>
<tr>
<td>Source of case study</td>
<td>Follow-up after researching the app’s repo on GitHub.</td>
</tr>
<tr>
<td>Catalyst for the case study</td>
<td>The developer was happy to support this research.</td>
</tr>
<tr>
<td>Approvals</td>
<td>Informal, directly from the creator of the project</td>
</tr>
<tr>
<td>Research methods</td>
<td>Online interview, email discussions, etc.</td>
</tr>
<tr>
<td>Participants</td>
<td>The main developer of the project</td>
</tr>
<tr>
<td>Analytics collected</td>
<td>Google Play Console with Android Vitals</td>
</tr>
<tr>
<td>Research software</td>
<td>None applicable.</td>
</tr>
<tr>
<td>Additional data collected</td>
<td>Interview notes and emails</td>
</tr>
<tr>
<td>Active period</td>
<td>July 2020</td>
</tr>
<tr>
<td>Post-hoc analysis</td>
<td>Analysis of updates to the source code</td>
</tr>
</tbody>
</table>

Data collection and use: included contemporaneous notes made during the interview, GitHub codebase history, and additional email follow-up discussions. The data sources: are summarised in Table 5.10.

Exodus Privacy confirms two analytics libraries are included: Firebase Analytics and Crashlytics https://reports.exodus-privacy.eu.org/en/reports/152278/.

Data analysed included:

- **Contemporaneous notes**: These were summarised, by me, and shared both by email and in discussion with co-researchers on the Firebase Analytics remote logging project. Note: the audio calls of the meetings with fellow researchers were performed in Zoom and generally recorded.

- **Emails**: Including corroboration that Google required Android developers who used Fabric Crashlytics SDK to migrate to the Firebase Crashlytics SDK by 15th November 2020.

- **Mobile analytics**: The usage and effects of mobile analytics were discussed however the contents were not provided as the app was no longer being developed actively.

- **Issues database**: https://github.com/Phantast/smartnavi/issues of which app crashes on moto defy discusses tradeoffs between releasing the app on F-Droid that prohibits Crashlytics and other commercial Mobile Analytics and the challenge of the developer knowing about, finding, and eventually being able to address the cause of the crash. In app crashes when launching the seemingly technical and competent author provided a partial crash log which enabled the bug to be identified and fixed by the app’s developer.

The project’s creator developed the app as part of his bachelors and masters degrees in Germany. He continued to develop and maintain it afterwards. The project does not have any automated tests, instead
he tests the app interactively. The project is unusual as, according to the creator, it provides a test bed for students at at least one German University researching Marketing, unfortunately I have yet to find public references to this.

**Corroboration:** a discussion with the interviewee on the analysis and findings.

---

**Table 5.10: SmartNavi: data sources**

<table>
<thead>
<tr>
<th>Data Source</th>
<th>Records</th>
<th>Volumes</th>
<th>Analysis method</th>
<th>Contribution</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Development artefacts</td>
<td>Sourcecode**</td>
<td>92 commits</td>
<td>Code analysis</td>
<td>Understanding their use of Firebase and Crashlytics</td>
<td></td>
</tr>
<tr>
<td>Issues database</td>
<td>smartnavi/issues</td>
<td>18</td>
<td>Sense-making</td>
<td>Insights into the project &amp; their use of mobile analytics</td>
<td>Online interview.</td>
</tr>
<tr>
<td>Pre-study interviews</td>
<td>contemporaneous notes</td>
<td>1</td>
<td>Ask the app devs</td>
<td>Discussion on the migration to Firebase Crashlytics</td>
<td></td>
</tr>
<tr>
<td>Mid-study communications with developers</td>
<td>GMail</td>
<td>10^3</td>
<td>Ask the app devs</td>
<td>Email conversations.</td>
<td></td>
</tr>
</tbody>
</table>

**https://github.com/Phantast/smartnavi**
5.6 App-centric: Kiwix

From a research perspective the purposes of this case study contributed multi-year perspective on the use of crash analytics by a hybrid development team (led by a professional developer) who actively attended to failures reported by the platform-level mobile analytics (Android Vitals). The project also facilitated a comparison between an experiment app and a control app which demonstrated the improvements in the reliability of the experiment [app] were most likely to be as a result of the release and use of that app, which had fixes applied to that app’s codebase. The self-imposed restrictions on their rejection of embedding any analytics in the app also gave the project added value from a research perspective as their only source of mobile analytics came from the platform.

**Case study participants**: included the two primary project leads. They were, and still are, the marketing/finances/business lead and the engineering lead. This was combined with ongoing ad-hoc discussions with various developers who worked on the Kiwix project in general and the Android app in particular. Overall, at least 10 of the project team participated in work directly related to this research.

The Kiwix project started as a way to make Wikipedia available offline, globally [198]. The team wrote software and implemented systems to do so and have worked closely with the WikiMedia Foundation for years. They also make vast amounts of other content available including StackOverflow and TED talks. Innumerable teams, projects, and people use Kiwix in various guises. Table 5.11 provides a succinct overview.

Intentionally the Kiwix project do not integrate mobile analytics in the apps to protect the safety and privacy of users; however, they do use the anonymous platform analytics provided by app stores, including Google Play Console.

The project has multiple opensource projects including software tools that download content from various sources including Wikipedia, web servers that serve content, and a wide range of desktop and mobile apps. I have been a part-time volunteer with Kiwix since 2014. I have contributed to several of their opensource projects including the Android app where I helped with automated testing and with continuous builds, etc.

One of the many benefits of the project’s openness is the visibility into the developers who have developed and maintained the source code. Many contributors joined as volunteers through Google Summer of Code [199] or Google Code-in [11]. Several became core contributors for a year or more, and some now work for leading technology businesses. There have been occasional contributions from Google software developers who volunteer their time. The codebase and development artefacts are all opensource, much of the communications is also publicly accessible, e.g. using GitHub issues, IRC and Slack [12]. In addition there are emails and informal discussions, etc.

The project uses free-to-use services, e.g. GitHub for the codebases. The Continuous Build service was Travis-CI at the time of the case study [13] and the project had a pro bono account on the commercial BitBar device testing farm [14]. The codebase included a mix of automated unit tests and

---

10: github.com/kiwix/kiwix-android/graphs/contributors
[199]: Google (2020), Google Summer of Code
11: codein.withgoogle.com/archive/
Note: Google Code-in was shutdown and the history archived by Google in 2020.
12: https://wiki.kiwix.org/wiki/Communication
13: Since replaced by GitHub Actions
14: https://bitbar.com/
app tests. In short, the project used fairly well honed tools and practices to manage their codebase, software contributions, perform code quality checks, and run the automated tests on both virtual and physical Android devices.

At the start of the case study there were two parallel releases in progress, the production 2.x release and a planned 3.0 release. **Data Sources:** for this case study are in Table 5.12.

**Interview format:** open-ended discussions with the project leads who asked for help to reduce the failure rate of the project’s Android apps. These were followed by *ad-hoc* interview discussions with both project leads and working discussions with the Android developers focused on their use of mobile analytics for crash analysis.

**Data collection and use:** The data collected included Android Vitals reports, contemporaneous notes from discussions, and hackathon summaries.

**Data analysed:** the majority of the data came from Android Vitals reports collected both interactively and using vitals scraper. These reports recorded the failures and failure rates at various levels of granularity. The issues database and the source code documented the changes to the project’s artefacts. The emails and notes recorded the thinking, rationale, and so on. They also included about ways the app crashed (e.g. from the testing performed by [http://www.test1080.com/](http://www.test1080.com/)). When the professional development lead joined (Apr 2019) and left the project (Dec 2020).

The **primary intervention** was for several of the team to address several of the most prevalent crashes during the Kiwix hackathon in August 2019, in Stockholm, Sweden. Follow on bug fixes and an increased interest in using mobile analytics outputs led to further improvements in the stability of the main Android app. When the custom apps were refreshed using the improved codebase their stability also increased.

### Table 5.11: Case Study key facts: Kiwix

<table>
<thead>
<tr>
<th>Website</th>
<th><a href="http://www.kiwix.org/en/">www.kiwix.org/en/</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>Google Play Home</td>
<td>play.google.com/store/apps/dev?id=9116215767541857492</td>
</tr>
<tr>
<td>Founded</td>
<td>2007</td>
</tr>
<tr>
<td>Business Domain</td>
<td>Education</td>
</tr>
<tr>
<td>Business type</td>
<td>Not for profit association</td>
</tr>
<tr>
<td>Technologies</td>
<td>Native platform apps And associated software tools.</td>
</tr>
<tr>
<td>Source code</td>
<td>Opensource <a href="https://github.com/kiwix">https://github.com/kiwix</a></td>
</tr>
<tr>
<td>Analytics used by</td>
<td>Google Play Console with Android Vitals.</td>
</tr>
<tr>
<td>Development Practices</td>
<td>Core developers combined with part-time volunteers.</td>
</tr>
<tr>
<td>User base</td>
<td>100,000’s for the Android app</td>
</tr>
<tr>
<td>Installations</td>
<td>1,000,000+ for the Android app</td>
</tr>
<tr>
<td>Source of case study</td>
<td>Long-term engagement with the project.</td>
</tr>
<tr>
<td>Catalyst for the case study</td>
<td>Excessively high crash rates for several key apps.</td>
</tr>
<tr>
<td>Approvals</td>
<td>Informal, full support of the project leads.</td>
</tr>
<tr>
<td>Intervention</td>
<td>Hackathon, and occasional contributions.</td>
</tr>
<tr>
<td>Research methods</td>
<td>Field Experiment, Hackathon, and see Table 5.12.</td>
</tr>
<tr>
<td>Participants</td>
<td>Approximately 10 people including both project leads</td>
</tr>
<tr>
<td>Analytics collected</td>
<td>Google Play Console with Android Vitals.</td>
</tr>
<tr>
<td>Research software</td>
<td>None applicable.</td>
</tr>
<tr>
<td>Additional data collected</td>
<td>Emails from Google Play Console, development artefacts.</td>
</tr>
<tr>
<td>Active period</td>
<td>March 2019 - March 2020</td>
</tr>
<tr>
<td>Post-hoc analysis</td>
<td>Ongoing analysis with recent re-engagement.</td>
</tr>
</tbody>
</table>
Table 5.12: Kiwix: data sources

<table>
<thead>
<tr>
<th>Data Source</th>
<th>Records</th>
<th>Volumes</th>
<th>Analysis method</th>
<th>Contribution</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Development artefacts</td>
<td>Sourcecode</td>
<td>$10^5$</td>
<td>Code analysis</td>
<td>History of commits with crash fixes Bugs</td>
<td></td>
</tr>
<tr>
<td>Development artefacts</td>
<td>Issues database</td>
<td>$10^5$</td>
<td>Artefact analysis</td>
<td>Holistic discussion of the hackathon. Reflections on the progress of this the first of the action research case studies</td>
<td>Email conversations</td>
</tr>
<tr>
<td>Pre-study interviews</td>
<td>GMail</td>
<td>$10^2$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Field notes</td>
<td>Contemporaneous notes</td>
<td>$10^2$</td>
<td>Observation and Analysis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Analytics tools &amp; artefacts</td>
<td>Interactive screenshots from Google Play Console with Android Vitals &amp; Vitals-scraper outputs</td>
<td>$10^2$</td>
<td>Sensemaking, ask the tool devs</td>
<td>Measured ongoing improvements. Outputs were discussed with Google Engineering.</td>
<td></td>
</tr>
</tbody>
</table>

Corroboration: the causes, fixes, and improvements in the failures were discussed on an ongoing basis with developers and with the project leads. A draft of my MobileSoft 2019 paper[2] was reviewed by Stephane, one of the project leads. It was also shared with the then lead of the Android project (Isaac) and both Emmanuel and Stephane.

5.7 App-centric: Catrobat

Why is this case study included? the project corroborated the improvements that applying mobile analytics could provide even teams who had implemented many other recommended software development practices. It also enabled comparisons between Fabric Crashlytics, the side-effects of migration to Firebase Crashlytics.

The case study participants are listed in descending order based on their level of participation. They are: the product owner (Matthias Müller) and the project lead (Prof. Wolfgang Slany), various developers of the mobile apps in a workshop in Graz, Austria the hackathon in Graz, and the participants of the workshop in Wroclaw, Poland, a total of approximately 20 people.

The Catrobat project was created and is actively developed by a team in the Graz University of Technology, Austria 15. It consists of the flagship Pocket Code app, several custom branded derivatives, and the increasingly popular Pocket Paint app which emerged from the Pocket Code app where it remains as a subset of the overall Pocket Code’s functionality. The project started in 2010, has had over 1,300 contributors, 4 million downloads, and 350 thousand active users, and is used in 180+ nations in 60+ languages [200].

The Android app in this case study is an extremely and unusually well researched and properly developed app and codebase. There are at least 216 contributors for the Android Pocket Code app [201].

The case study included two main events, 1) a hackathon in November 2019 and 2) participation in a pre-conference workshop in Poland in February 2020. We agreed on a hackathon for a couple of reasons: Kiwix
5 Overview of the case studies

Table 5.13: Case Study key facts: Catrobat

<table>
<thead>
<tr>
<th>Website</th>
<th><a href="https://catrobat.org/">https://catrobat.org/</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>Founded</td>
<td>2010</td>
</tr>
<tr>
<td>Business Domain</td>
<td>Education &amp; Visual programming.</td>
</tr>
<tr>
<td>Business type</td>
<td>Not for profit association.</td>
</tr>
<tr>
<td>Technologies</td>
<td>Android Jenkins CI <a href="https://jenkins.catrob.at/job/Catroid/">https://jenkins.catrob.at/job/Catroid/</a> JIRA <a href="https://jira.catrob.at/">https://jira.catrob.at/</a></td>
</tr>
<tr>
<td>Source code</td>
<td>Opensource <a href="https://github.com/Catrobat">https://github.com/Catrobat</a></td>
</tr>
<tr>
<td>Analytics used by team</td>
<td>Fabric Crashlytics, Google Play Console</td>
</tr>
<tr>
<td>Development Practices</td>
<td>Sophisticated (see text).</td>
</tr>
<tr>
<td>User base</td>
<td>100,000’s for the Android app</td>
</tr>
<tr>
<td>Installations</td>
<td>1,000,000+ for the Android app</td>
</tr>
<tr>
<td>Source of case study</td>
<td>Discussion at MobileSOFT 2019 conference.</td>
</tr>
<tr>
<td>Catalyst for the case study</td>
<td>Excessively high crash rates for their flagship app.</td>
</tr>
<tr>
<td>Approvals</td>
<td>Their head of department with full support of the project leads.</td>
</tr>
<tr>
<td>Intervention</td>
<td>Hackathon, pre-conference workshop.</td>
</tr>
<tr>
<td>Research methods</td>
<td>Hackathon, online interviews, email discussions, etc., and see Table 5.14.</td>
</tr>
<tr>
<td>Participants</td>
<td>20 people including the product owner and product lead, app developers, workshop participants</td>
</tr>
<tr>
<td>Analytics collected</td>
<td>Fabric Crashlytics, Google Play Console with Android Vitals</td>
</tr>
<tr>
<td>Research software</td>
<td>None applicable.</td>
</tr>
<tr>
<td>Additional data collected</td>
<td>Interview notes and emails</td>
</tr>
<tr>
<td>Post-hoc analysis</td>
<td>Ongoing access to Google Play Console with Android Vitals and the development artefacts.</td>
</tr>
</tbody>
</table>

found them beneficial and productive, and the Catrobat team wanted to have a short, unusual and interesting way to try out the concept of using mobile analytics outputs to improve reliability that would also appeal to their developers.

Their project leads selected Pocket Code as the app we would use for the field experiment as it had the higher crash rate and was also a significantly more complex app than their other core app Pocket Paint which was relatively self-contained and simple in terms of both functionality and codebase.

A second workshop was planned for 28th Feb 2020. The preparation included opensourced course materials. However in the day the workshop did not go as envisaged because of the outbreak of Covid-19 that weekend in parts of Europe. Subsequently the project team in Austria was no longer available because of the effects of the pandemic which adversely affected the planned collaboration.

The development microcosm was sophisticated and one of the most mature in terms of opensource mobile app ecosystems.

### Ranking Catrobat’s development practices

Catrobat is one of a small set of projects who incorporate all the recommended practices. Here are comparisons with two related research papers.

**Automated tests**: In an admittedly small sample only 9 of 19 opensource Android app projects had any automated tests. And in a larger body of research 40.6% of 1000 projects have automated tests [137, p. 2461].

---

16: github.com/julianharty/testing-with-analytics-workshop/

17: I have worked in opensource for 15 years, including at Google, ebay, and other organisations so I say this based on my professional experience.

[202]: Bernardo Silva et al. (2016), ‘An analysis of automated tests for mobile Android applications’

[137]: Cruz et al. (2019), ‘To the attention of mobile software developers: guess what, test your app!’
Table 5.14: Catrobat: data sources

<table>
<thead>
<tr>
<th>Data Source</th>
<th>Records</th>
<th>Volumes</th>
<th>Analysis method</th>
<th>Contribution</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-study interviews</td>
<td>contemporaneous notes</td>
<td>3</td>
<td>Ask the app devs</td>
<td>Set initial context, the baseline, and scope case-study related updates and planning</td>
<td></td>
</tr>
<tr>
<td>Mid-study communications with developers</td>
<td>GMail</td>
<td>10³</td>
<td>Ask the app devs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Analytics tools &amp; artefacts</td>
<td>Interactive screenshots &amp; Vitals-scraper outputs</td>
<td>10³</td>
<td>Sensemaking</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Development artefacts</td>
<td>Issues database, and Jenkins CB Dashboard</td>
<td>20⁺</td>
<td>Observation and Analysis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Field notes</td>
<td>various</td>
<td>10³</td>
<td>Observation and Analysis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hackathon development artefacts</td>
<td>various, including: the issues raised during the hackathon</td>
<td>10³</td>
<td>Sense-building, sense-making</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CI/CD and automated tests: The Catrobat project has automated tests and CI/CD, only 14.7% of 1000 open source Android apps do so [p. 2461] ¹⁸

The Jenkins builds measure various forms of code coverage and the results are public: jenkins.catrobat.at/job/Catroid/job/develop/.

Promoting good practices: Furthermore, Cruz et al found “only 19 [of the 1000 projects] are actually promoting full test coverage with coverage tracking services” [p. 2462]. The Catrobat project used code quality tools and aims for zero warnings from these tools. They also developed their own custom test automation framework and integrated Fabric Crashlytics into their flagship Pocket Code Android app.

The Android app in this case study is an extremely well researched and properly developed app and codebase. The project started in 2010, has had over 1,300 contributors, 4 million downloads, and 350 thousand active users, and is used in 180+ nations in 60+ languages [200]. There are at least 216 contributors for the Android Pocket Code app [201].

Many perceived good practices were and are assiduously applied on an ongoing basis, for instance: Test-Driven Development, Clean Code [203], a documented consistent Workflow and Pull Requests, and Continuous Integration. The codebase is far more complex than the Kiwix Android apps and the app is significantly richer in terms of the features and functionality [204].

Interview format: started with open-ended interviews were used to learn of their concerns, their use of mobile analytics, and their pain-points.

Interview conducted: An open discussion with the product owner; which led to a more structured discussion with the project lead who selected the app to focus on. During both hackathons there were ad-hoc conversations with the developers and other participants (who were predominantly professional software testers).

Catrobat: Interventions: The key intervention was organise a weekend hackathon with an open invitation for any of the extended development team to participate. During day 1 of the hackathon, after informal introductions and a discussion about the aims of the hackathon, the

¹⁸: Interestingly Cruz et al did not evaluate either Pocket Code or Pocket Paint. They also discounted ‘self-hosted’ CI including Jenkins which Catrobat uses extensively.

[200]: Catrobat (2021), Home - Catrobat
[201]: Catrobat Team (2021), Catrobat/-Catroid: Writing programs on an Android device without prior knowledge.
[203]: The Catrobat Project (2015), First steps into Catrobat
next task was to create tickets in JIRA for the top 10 crash clusters and the top 10 ANR clusters as reported by Android Vitals. These were reported in JIRA during the first hour of the hackathon, the complete set are available online at jira.catrob.at/browse/CATROID-4187?jql=labels: hackathon-2019.

The participants, in ones or twos, selected one of these tickets and worked on it. They then selected another ticket and worked on that one. They continued for approximately 5 hours until late afternoon that day. The event closed with a communal meal at a local pizzeria. The participants chose not to continue with day 2 of the hackathon (which was on the Sunday), instead they preferred to work on the issues during the normal working week (Monday to Friday). Several of them did so and continued to work on various tickets raised in the hackathon. The project team made two related releases of the Pocket Code Android app, with cumulative fixes in these releases.

**Data collection and use:** Data was collected from all the mobile analytics sources: Fabric Crashlytics, Firebase Crashlytics, Google Play Console and Android Vitals. Data was also collected from the project’s github codebases, issues database, project wiki, and Jenkins, their continuous integration service. The remaining sources of data included contemporaneous notes and emails. A summary of the data sources are in Table 5.14.

Examples of contributions from the various data sources include:\[^{19}\]

- **Issues database:** Recorded the top 10 crashes and top 10 ANRs identified during the hackathon and measured their progress as some of them were addressed.
- **Emails:** Early snapshots of the statistics for their two main Android apps from Google Play Console and Android Vitals. Sources of additional information, for example about correlations with higher crash rates and Huawei phones [https://jira.catrob.at/browse/CATROID-373](https://jira.catrob.at/browse/CATROID-373), about the reasons the project team wanted to have a hackathon (to reduce the measured crash rate from 4% to below the bad behavior threshold of 1.09%), details of who was invited to the first hackathon and who participated (6 developers - of these 2 were students and 4 were longer-term contributors who had completed their academic studies, Matthias, Wolfgang, Joe, me). Releases of vitals scraper shared with the project leads. Confirmation that crashlytics reporting was broken in the Android release shortly before 17th November 2019. Records of progress in improving the crash rate of Pocket Code. Confirmation of the bug fix release, released on 6th Jan 2020. Utility of a Turkish review that pinpointed a long standing bug that was fixed server-side around 1st April 2020.
- **Mobile analytics:** Identified vast differences in the counts reported by Fabric Crashlytics and Android Vitals; measured the cumulative improvements in the app at the time of the hackathon and for two months afterwards.

**Corroboration:** The results and outcomes of the workshops and follow-up bug fixing were reviewed with the product owner and the product lead. Some of the initial results were published in a joint paper with Matthias Mueller[3].

---

[^{19}]: NB: Several of these may fit better in the findings chapters.

[3]: Harty et al. (2019), ‘Better Android Apps Using Android Vitals’
5.8 App-centric: C1

Case study C1 is based on a project at a multi-billion dollar international business outside the USA. Details of the project are confidential and therefore not published in this thesis, nonetheless various findings can be shared as part of this research.

<table>
<thead>
<tr>
<th>Website</th>
<th>Confidential</th>
</tr>
</thead>
<tbody>
<tr>
<td>Google Play Home</td>
<td>Confidential</td>
</tr>
<tr>
<td>Business Domain</td>
<td>A high tech corporation</td>
</tr>
<tr>
<td>Business type</td>
<td>An international company</td>
</tr>
<tr>
<td>Technologies</td>
<td>Confidential</td>
</tr>
<tr>
<td>Source code</td>
<td>Closed</td>
</tr>
<tr>
<td>Analytics used by team</td>
<td>Microsoft App Center, other commercial products, proprietary code, and Google Play Console.</td>
</tr>
</tbody>
</table>

**Table 5.15: Case Study key facts: C1**

<table>
<thead>
<tr>
<th>Development Practices</th>
<th>Multiple teams working on the Android app.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Installations</td>
<td>1,000,000’s for the Android app</td>
</tr>
<tr>
<td>Source of case study</td>
<td>Commercial engagement</td>
</tr>
<tr>
<td>Catalyst for the case study</td>
<td>The industry project wished to improve the quality of their products and service to their user base.</td>
</tr>
<tr>
<td>Approvals</td>
<td>Documented in the commercial Master Services Agreement.</td>
</tr>
<tr>
<td>Interventions</td>
<td>Consultant with several of the development teams and their engineering leadership.</td>
</tr>
<tr>
<td>Research methods</td>
<td>Various</td>
</tr>
<tr>
<td>Participants</td>
<td>CEO and CTO of entire business, CTO of the project, approximately 17 members of the development team</td>
</tr>
<tr>
<td>Research software</td>
<td>Microsoft App Center, Google Play Console with Android Vitals</td>
</tr>
<tr>
<td>Additional data collected</td>
<td>Additional Analytics and logs, details are confidential.</td>
</tr>
<tr>
<td>Active period</td>
<td>Q4 2020 - Q2 2021</td>
</tr>
<tr>
<td>Post-hoc analysis</td>
<td>Access to the materials was available after the action research stage and evaluated both for the project team and for research purposes. A summary of the findings and results were provided to senior management; and the findings and the results achieved were discussed with one of the engineering leadership team.</td>
</tr>
</tbody>
</table>

The researcher accepted a consulting engagement with the corporation and was asked to assist one of their key projects. This project included an Android app, online APIs developed by the larger project team, and other apps, systems and services. It also incorporated other internal systems, APIs, and services provided by other development teams.

The case study incorporated a very popular complex mobile app with real-time performance requirements developed by an extremely large development team (larger than any of the other app-centric case studies by an order of magnitude). The first that included both proprietary and commercial mobile analytics SDKs.

**Case study participants:** were approximately 20 people. These included the CTO of the project, and his managers (2 people known as Apex management who were effectively the CEO and CTO of the entire organisation of between 6,000 and 10,000 people), the Android developers, their development leads, and their managers; several of the consulting company Xnsio (also known as Znsio).

Owing to confidentiality and other contractual obligations details have been removed from this case study. Several practical representative
examples have been made available as opensource projects so they can be independently corroborated.

The overall project team comprised over 100 people working directly on the product. Developers worked in a matrixed organisation [205]. Multiple groups of developers worked on the Android app.

**Interview format:** was open-ended for the introductory interviews/discussions. These were followed with thematic for interviews with developers, leads, and their manager.

**Interviews conducted:** Initially the interviews were led by the business so they could determine the suitability of being involved in the research, they then switched to the researcher leading the interviews once the case study started. Generally the interviews were conducted using video conferencing, several were complemented by written discussions in Microsoft Teams and/or emails.

**Data collection and use:** The data collected during the case study included computer-generated reports and other outputs from various of the mobile analytics services, source code, automated Continuous Integration (CI) and Continuous Build (CB) results, issues database, email and Microsoft Teams conversations and updates, contemporaneous notes. Correspondence with a commercial provider of mobile analytics about their export mechanisms.

The data used in this research was restricted to only non-identifiable materials; the rest of the material has corroborated and aligns with findings from other case studies covered in this thesis.

The data was collected contemporaneously during the consulting engagement and subsequently. Details cannot be provided here nonetheless the methods described in the Methodology chapter were used from a research perspective. Field notes were made contemporaneously in addition to contributions to development artefacts.

**Intervention:** Working with several teams, including those working on the Android app in particular, to reduce the crash and ANR rates for the app beyond a phased two-stage set of improvements (which were achieved using a combination of Google Play Console with Android Vitals as the primary source and measure and Microsoft App Center to augment, corroborate, and provide inter-tool-comparisons).

**Corroboration:** the findings and the overall results were reviewed with the Apex management and the head of the consulting company who represented the rest of the engineering team.

### 5.9 Augmenting the app-centric case studies: field experiments

The app-centric case studies provided various vectors into a rich and complex area, however as they were all for real world apps in production there were many aspects that did not occur during any of these case studies. During the research there were various field experiments incorporated into the research. These are broadly in three areas:
1. Creating small Micro experiments in the form of Android apps that generally exercise an in-app mobile analytics service in addition to whatever functionality they provide (See 5.9.1).
2. Contributing to opensource mobile analytics projects (Section 5.9.2).
3. Research into logging performed by Android app developers and the development of software utilities to further that research (Section 5.9.3).

These are followed by research into analysis of the source code of 107 Android apps to learn how developers use Firebase Analytics, more details in Section 5.10.

5.9.1 Experimental apps to increase coverage of mobile analytics services

Various tests were performed to augment the app-centric case studies. The tests were packaged as several, small, experimental Android apps; they are not currently intended to go into general release.

- Travel Europe: a pre-release app that incorporated content from Wikipedia and in-app mobile analytics from Microsoft’s App Center.
- zipternet: https://github.com/ISNIT0/zipternet incorporated Microsoft App Center.
- Iteratively Demo and Test Analytics (IDoT): https://github.com/commercetest/idot (access currently available upon request) incorporated the Iteratively SDK to facilitate testing and evaluating the capabilities it provided.
- AndroidCrashDummy: https://github.com/ISNIT0/AndroidCrashDummy, used in the log-centric experiments (see Section 5.9.3. Android log-centric experiments).

5.9.2 Contributions to opensource mobile analytics projects

Projects that are opensourced are not necessarily easy to contribute to, there may be various hurdles and sufficient delays to dissuade the majority of developers from actually contributing to those projects. These field experiments were opportunistic in that they emerged as part of the reset of the research, nonetheless they are realistic as they included real-world improvements to those mobile analytics projects and the respective development teams reviewed and approved the contributions.

- PostHog: Two commits to improve the documentation
- Sentry: Improved the onboarding documentation

**PostHog:** the research included two commits to their codebase, both improvements to their documentation. They use a an automated ‘bot’, netlify, that checked the changes and one of their employees approved and merged the changes. A nice gesture was another of their ‘bot’s added
20: A set of three branded mugs and a sticker.

me as a contributor to their project, Add julianharty as a contributor, and they provided a code to obtain some ‘merch’.

**Sentry**: the research consisted of a single commit to their codebase, to improve their developer-oriented documentation. They also use an automated ‘bot’, Vercel, that reviewed the changes and eventually deployed the changes once the code review was completed successfully. The code-review included two of their team where we discussed and refined the change before they accepted it.

In both cases the companies were professional, friendly, and keen to improve the quality of their product and documentation. They provided various mechanisms to help ensure contributions were of sufficient quality and the people involved engaged in discussions about how they worked, etc. Both provide clearly visible encouragements for people to participate and contribute.

### 5.9.3 Android log-centric experiments

Mobile analytics provides mechanisms developers can use to perform implicit and explicit logging where the logs are generated at runtime when the app is being used. This section provides an overview of the log-centric experiments and their related opensource projects; and the next section includes analysis of 107 opensource Android apps that use Firebase Analytics for logging. (Section 5.10. Augmenting the app-centric case studies: sourcecode analysis).

The experiments extracted Android logging statements and then analysed them. This work was complemented with creating Assert statements that automated tests could use to check whether expected log messages had been emitted into the Android log.

The related opensource projects are:

- Automated tests for Android log messages: [github.com/ISNIT0/AndroidLogAssert](https://github.com/ISNIT0/AndroidLogAssert)
- Log Searcher: [github.com/ISNIT0/log-searcher](https://github.com/ISNIT0/log-searcher)
- Logcat filter and analysis tool: [github.com/ISNIT0/logcat-filter](https://github.com/ISNIT0/logcat-filter)
- Log complexity comparison: [github.com/ISNIT0/log-complexity-comparison](https://github.com/ISNIT0/log-complexity-comparison)

### 5.10 Augmenting the app-centric case studies:

sourcecode analysis

Sections 5.1 to 5.4 presents four app-centric case studies where the developers were asked of their experiences of using mobile analytics. None of these provided access to the source code of their apps. In contrast 5.5) to 5.8) did provide access to their sourcecode. As only three of these case studies have opensourced their source code, and only SmartNavi currently uses an in-app mobile analytics SDK the app-centric case
studies were complemented that research by investigating the source code of 107 open source projects for active Android apps where the source code was freely available on GitHub.

What these 107 projects had in common was they used recent releases of Firebase Analytics. 50 of these simply initialised the SDK and did not include any custom calls to the SDK, the remaining 57 did make custom calls. The analysis of the code was jointly performed with an international group of researchers and published in Harty et al. [9]. One of these 57 projects, Smartnavi (see Section 5.5), also became an app-centric case study in that the developer was interviewed about their use of mobile analytics.

<table>
<thead>
<tr>
<th>Website</th>
<th><a href="https://fabric.io/">https://fabric.io/</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>Founded</td>
<td>2012</td>
</tr>
<tr>
<td>Business Domain</td>
<td>Crash reporting and analytics for mobile apps.</td>
</tr>
<tr>
<td>Source code</td>
<td>A subset is open source <a href="https://firebaseopensource.com/projects/firebase/firebase-android-sdk/readme/">https://firebaseopensource.com/projects/firebase/firebase-android-sdk/readme</a></td>
</tr>
<tr>
<td>User base</td>
<td>1,000,000’s of app developers for their analytics (see text)</td>
</tr>
<tr>
<td>Installations</td>
<td>Billions for the Android analytics</td>
</tr>
<tr>
<td>Research methods</td>
<td>Grey Data, Grey Literature, analytics tools &amp; artefacts, field notes.</td>
</tr>
<tr>
<td>Analytics collected</td>
<td>Fabric Crashlytics and Firebase Crashlytics reports.</td>
</tr>
<tr>
<td>Research software</td>
<td>None applicable.</td>
</tr>
<tr>
<td>Additional data collected</td>
<td>N/A.</td>
</tr>
<tr>
<td>Active period</td>
<td>Q3 2019 to Q2 2020.</td>
</tr>
<tr>
<td>Relevant app-centric case studies</td>
<td>Catrobat, Smartnavi.</td>
</tr>
</tbody>
</table>

### 5.11 Tool-centric: Crashlytics

Crashlytics grew from a group of developers who wanted to scratch an itch [206] into a product that first Twitter and then Google acquired as it became increasingly popular [207]. Like many projects and products it morphed over the years and in 2020 Google completed the integration of Crashlytics into Firebase and removed support for older versions of the SDK (which had some knock-on effects in terms of privacy and what the reports contain). Table 5.16 provides an overview of Crashlytics.

The user base is based on a claim in a blog post written by the founder [208] with the assumptions: Android apps were a significant portion, the mean ratio of apps to developers is less than 5, the overall use has increased since 2015. Similar estimates are used to determine the installations from another blog post by the company’s founder [209]. The true figures are hard to ascertain as Google have integrated Crashlytics into Firebase.

The primary source of this case study is via the Catrobat case study where the development team were using Fabric Crashlytics in their flagship Pocket Code Android app. Crashlytics also surfaced in several of the developer interviews, and in source code for various open source projects. Furthermore the researcher has been aware of Crashlytics through his professional work since at least 2015.

**Crashlytics: Data collected and methods used for collection** The majority of the data collected was done so interactively through using the various reporting user interfaces provided by Fabric and Firebase. Additional
data was collected though Grey Data and Grey Literature searches and through analysis of development artefacts.

## 5.12 Tool-centric: Firebase Analytics

Firebase was launched in 2012[^1] and Firebase Analytics was launched in 2016[^2]. Various industry sources concur that Firebase Analytics is the most popular mobile analytics library and in well over 50% of Android apps, for example, 55% of the Android apps it has analysed have Firebase Analytics[^2], while AppBrain states it was in 82.88% of Android apps and an astonishing 99.58% of installs of new apps[^2].

Table 5.17 provides an overview of the key facts for Firebase Analytics.

The case study emerged from developer interviews as part of several of the app-centric case studies. Subsequent joint research was performed in 2020 to analyse 107 opensource Android apps on GitHub that included Firebase Analytics in their codebase[^9].

**Firebase Analytics: Data collected and methods used for collection**

Screenshots provided by developers and sent to the researcher by email. Source code was cloned from the respective opensource repos on GitHub and analysed using several tools including Google Sheets, Android Studio, and srcML (detailed in Harty et al.[^9]).

---

[^1]: Harty et al. (2021), ‘Logging Practices with Mobile Analytics: An Empirical Study on Firebase’

[^9]: Extrapolated from ‘82.88% of installs’[^2] and ‘over 2.5 billion active Android devices’[^2] and assuming there are at least the same quantity of active Android devices in 2022.

---

**Table 5.17: Tool Centric Case Study key facts: Firebase Analytics**

<table>
<thead>
<tr>
<th>Website</th>
<th><a href="https://firebase.google.com/products/analytics">https://firebase.google.com/products/analytics</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>Founded</td>
<td>2016</td>
</tr>
<tr>
<td>Business Domain</td>
<td>Product analytics.</td>
</tr>
<tr>
<td>User base</td>
<td>1,000,000’s for their analytics</td>
</tr>
<tr>
<td>Installations</td>
<td>2,000,000,000+ for the Android analytics[^4]</td>
</tr>
<tr>
<td>Research methods</td>
<td>Ask the app devs, source code analysis, observation and analysis, sensemaking.</td>
</tr>
<tr>
<td>Analytics collected</td>
<td>Google Play Console with Android Vitals</td>
</tr>
<tr>
<td>Research software</td>
<td>None applicable.</td>
</tr>
<tr>
<td>Additional data collected</td>
<td>Interview notes and emails.</td>
</tr>
<tr>
<td>Active period</td>
<td>2019 to 2020</td>
</tr>
<tr>
<td>Relevant app-centric case studies</td>
<td>Moodspace, Moonpig, Smartnavi.</td>
</tr>
</tbody>
</table>
5.13 Tool-centric: Google Play Console with Android Vitals

Google Play Console incorporating Android Vitals is probably the largest composite source of analytics for mobile apps on Earth as it reports on up to several billion Android devices for the apps on that device.

Some history In 2010, Google announced an service that appears to be the first version of what became Android Vitals where users could submit crash reports that developers would receive in their Android Market account. Google continued to evolve their analytics which included providing developers a mechanism to download crash and ANR reports from 2015 until 2018. Google launched Android Vitals in 2017.

The research into Google Play Console with Android Vitals predates the app-centric case studies. It is one of the de-facto mobile analytics tools available to Android developers who release their apps in the Google Play ecosystem. Each of the app-centric case studies provided material related to this service that Google provides free of additional charge.

Google Play Console with Android Vitals: Data collected and methods used for collection The data included screenshots, extracts, and rekeying of contents from the interactive use of the mobile analytics service, interactive downloads of various monthly reports provided by the service, automated collection of screenshots and textual content through the use of Vitals Scraper, and material provided by interviewees who emailed screenshots and provided extracts. Source code was located through various searches online and through Grey Data, particularly through StackOverflow.

<table>
<thead>
<tr>
<th>Table 5.18: Tool Centric Case Study key facts: Google Play Console with Android Vitals</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Website</strong></td>
</tr>
<tr>
<td><strong>Service origin</strong></td>
</tr>
<tr>
<td><strong>Business Domain</strong></td>
</tr>
<tr>
<td><strong>Business type</strong></td>
</tr>
<tr>
<td><strong>Technologies</strong></td>
</tr>
<tr>
<td><strong>Source code</strong></td>
</tr>
<tr>
<td><strong>User base</strong></td>
</tr>
<tr>
<td><strong>Installations</strong></td>
</tr>
<tr>
<td><strong>Research methods</strong></td>
</tr>
<tr>
<td><strong>Analytics collected</strong></td>
</tr>
<tr>
<td><strong>Research software</strong></td>
</tr>
<tr>
<td><strong>Additional data collected</strong></td>
</tr>
<tr>
<td><strong>Active period</strong></td>
</tr>
<tr>
<td><strong>Relevant app-centric case studies</strong></td>
</tr>
</tbody>
</table>
5.14 Tool-centric: Iteratively with Amplitude

Two founders jointly created Iteratively and raised seed funding for the startup. Their focus was to develop tools and approaches to help product managers, developers, and analysts to have a single coherent and trustworthy data analytics pipeline from design to use. The startup was acquired by Amplitude who successfully listed on Nasdaq several months later, in 2021.

Iteratively’s CEO, contacted the researcher in May 2020. After an opening call with both founders, they were keen to share their experiences and make their tools and research available. The CEO confirmed both verbally and by email I was free to reuse their materials including for research purposes. They also accepted contributions to their materials and research, these were freely given and without charge or obligation. I introduced them to someone I worked with who they subsequently hired and they gave both of us permission to freely discuss details of their products, software, and research, again this is without charge or obligation, nonetheless there is an implicit moral obligation to protect sensitive material so shared and which is being upheld during this research.

The key facts for Iteratively with Amplitude are presented in Table 5.19.

Iteratively with Amplitude: Data collected and methods used for collection Field notes were recorded contemporaneously during the various calls and discussions with the founder and with the developer, in addition there are various email (and WhatsApp) discussions. Field notes were also made, together with screenshots, when using the tools and their online reports. The source code for the Android app is also a form of data, and is on GitHub. It is available: currently on request. It will be opensourced at github.com/commercetest/idot. Google Play Console’s Pre-launch reports have also been collected as screenshots using a web browser.
5.15 Tool-centric: Microsoft App Center

App Center combines various tools and utilities and includes in-app mobile analytics and crash reporting. Table 5.20 presents the key facts for this case study.

The crash reporting aspect probably started as part of the HockeyApp SDK. Microsoft acquired HockeyApp in late 2016 ‡‡ and integrated it into Microsoft App Center.

The earliest mention of Crash reporting in HockeyApp’s opensource Android SDK is in 2014 §§

A related opensource project, github.com/bitstadium/CrashProbe, and a related website called crashprobe.com 25, were both created by the team that developed HockeyApp. They compared the performance of various crash reporting SDKs. As it was opensourced developers of several of these SDKs contributed both source code and results. An example of the report from April 2017 has been cached by the web archive (www.crashprobe.com/ios/) and compares the results of 6 iOS crashreporting tools. Of note, their SDK only came third in the test results, which indicates the project was an honest contribution to improving the crash reporting SDKs. Microsoft preserved the project and the website post-acquisition but it fell into discuss and was eventually retired.

Estimates of the user base and installations: As many of the customers are likely to be corporations with large development teams, the ratio is likely to be several developers have access to AppCenter per app. According to AppBrain the App Center SDK has been installed in over 10 thousand apps and those apps have been downloaded over 3 billion times by 25th November 2020 and was in over 13 thousand apps and 30 billion downloads by 16th July 2021. It is in 8th place


§§ github.com/bitstadium/HockeySDK-Android/commit/3ccd53e44da791806720604b02d358de66ecbf6a.

---

Table 5.19: Tool Centric Case Study key facts: Iteratively with Amplitude

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Founded</td>
<td>2019</td>
</tr>
<tr>
<td>Business Domain</td>
<td>Software and services to help development teams capture clean, useful data to generate business insights.</td>
</tr>
<tr>
<td>Business type</td>
<td>Startup, since acquired, now part of Amplitude, that went Public in 2021.</td>
</tr>
<tr>
<td>Technologies</td>
<td>SDK generation tools Build and Integration tools, data validation tools for software analytics libraries.</td>
</tr>
<tr>
<td>Source code</td>
<td>Closed and not available for research.</td>
</tr>
<tr>
<td>User base</td>
<td>101 companies used their analytics pre acquisition, unknown post integration.</td>
</tr>
<tr>
<td>Installations</td>
<td>Not available.</td>
</tr>
<tr>
<td>Research methods</td>
<td>In person interviews, Grey Literature, email discussions, analysis of their Android SDK binary, and field experiment which led to observation and analysis.</td>
</tr>
<tr>
<td>Analytics collected</td>
<td>Screenshots of the Iteratively service in use and of Amplitude.</td>
</tr>
<tr>
<td>Research software</td>
<td>None applicable.</td>
</tr>
<tr>
<td>Additional data collected</td>
<td>Discussions with one of their development team.</td>
</tr>
<tr>
<td>Active period</td>
<td>May 2020 to 2022</td>
</tr>
<tr>
<td>Relevant app-centric case studies</td>
<td>None, it is part of the testing mobile analytics research.</td>
</tr>
</tbody>
</table>

Table 5.28: Tool Centric Case Study key facts: Microsoft App Center

<table>
<thead>
<tr>
<th>Website</th>
<th><a href="https://appcenter.ms/">https://appcenter.ms/</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>Launched as</td>
<td>Microsoft App Center</td>
</tr>
<tr>
<td>Business Domain</td>
<td>2017</td>
</tr>
<tr>
<td>Source code for the</td>
<td>IDE and related online</td>
</tr>
<tr>
<td>online service</td>
<td>services that includes</td>
</tr>
<tr>
<td>Source code for the</td>
<td>free and paid for</td>
</tr>
<tr>
<td>Android SDK</td>
<td>services.</td>
</tr>
<tr>
<td>Source code for a</td>
<td>Closed and not</td>
</tr>
<tr>
<td>sample Android app</td>
<td>available for research</td>
</tr>
<tr>
<td>And other repositories listed at</td>
<td><a href="https://github.com/Microsoft/appcenter/wiki/Repositories">https://github.com/Microsoft/appcenter/wiki/Repositories</a></td>
</tr>
</tbody>
</table>

| User base             | $10^4$ developers estimated to use their analytics (see text) |
| Installations         | 1,000,000,000+ for the Android analytics |

| Research methods      | None applicable. |
| Analytics collected   | March 2019 to Q2 2021 |
| Research software     | Cl; and it was also used in one of the small experimental apps (see Section 5.9.1. Experimental apps to increase coverage of mobile analytics services). |
| Additional data       | |
| collected             | |
| Active period         | |
| Relevant app-centric  | |
| case studies          | |

Microsoft App Center was integrated into several field experiment Android apps in 2019. It was also one of the main mobile analytics tools used in the commercial app-centric case study which extended the initial findings from the field experiments into mission-critical and real-world use of the service at scale based on an active end-userbase of millions of people.

Microsoft App Center: Data collected and methods used for collection

Screenshots and data obtained through the reporting APIs were collected as part of the assignment. In tandem field notes, email, and other collaboration tools were used on an ongoing basis during the action research period. Issues reported by others were also observed and analysed, for example in terms of how the issues were tracked, the changes to the app source code, and the subsequent testing of the potential fixes.

Grey Literature describes using Hockey App for crash reporting in iOS [215], the concepts of how to integrate and test crash reporting for Android was similar.

5.16 Tool-centric: Sentry

Sentry is a a good example of a mature investment-funded and purpose-built company who provide a mobile analytics platform together with various analytics services to their customers. They are one of several who have made their software freely available as opensource (others include Count.ly, PostHog, and Segment). Developers incorporate Sentry’s SDKs
Table 5.21: Tool Centric Case Study key facts: Sentry

<table>
<thead>
<tr>
<th>Website</th>
<th><a href="https://sentry.io/welcome/">https://sentry.io/welcome/</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>Founded</td>
<td>2011</td>
</tr>
<tr>
<td>Business Domain</td>
<td>Software Performance Analytics</td>
</tr>
<tr>
<td>Business type</td>
<td>Opensource code and paid for hosted services</td>
</tr>
<tr>
<td>Technologies</td>
<td>Multi-platform, Client and Server codebases.</td>
</tr>
<tr>
<td>Source code</td>
<td>Opensource <a href="https://github.com/getsentry">https://github.com/getsentry</a></td>
</tr>
<tr>
<td>User base</td>
<td>1,000,000’s developers use their analytics [216]</td>
</tr>
<tr>
<td>Installations</td>
<td>≥ 1% for the Android analytics¶¶</td>
</tr>
<tr>
<td>Research methods</td>
<td>Observation and Analysis, Field Experiment.</td>
</tr>
<tr>
<td>Analytics collected</td>
<td>Interactive reports and automated emails.</td>
</tr>
<tr>
<td>Research software</td>
<td>None applicable.</td>
</tr>
<tr>
<td>Additional data collected</td>
<td>Interview notes and emails.</td>
</tr>
<tr>
<td>Active period</td>
<td>Jan 2020 to Nov 2021</td>
</tr>
<tr>
<td>Relevant app-centric case studies</td>
<td>Local Halo; and it is part of the testing mobile analytics research.</td>
</tr>
</tbody>
</table>

and/or APIs into their software and can either self-host the necessary server-side software or use the services provided by Sentry. There is a free tier which has basic capabilities, and paid-for options. Table 5.21 presents the key facts for this case study.

One of the app centric case studies, LocalHalo (Section 5.2. App-centric: LocalHalo, uses Sentry and they provided access to their Sentry account. The access continued after the action research aspect of that case study and provided an ongoing view of the usage and the behaviours of that app, and the project’s website, until November 2021 when pricing changes by Sentry meant my access ceased.

Sentry: Data collected and methods used for collection

Data was collected interactively, via automated emails, and using their API service.

5.17 Miscellaneous sources

These include an opensource project, EduVPN, and six more mobile analytics software tools and associated services: AppPulse Mobile, AppSee, Azetone, Count.ly, MixPanel, PostHog, and Segment. For the first three of these I worked indirectly with the development team, evaluated the product and service and provided bug reports and other feedback. These engagements were part of my consulting work with and for what became HP Enterprise (and subsequently was acquired by MicroFocus). Some material has been published in Harty and Aymer [217].

- AppPulse Mobile: an innovative mobile analytics service that automatically instrumented Android apps during the build process of that app. The service provided crash and performance reporting. It was developed by HP, it is now owned by MicroFocus.
- AppSee: the first of two ‘heatmapping’ tools that included mobile analytics of the data collected by their in-app library. The company and their products were acquired and are no longer available.

[217]: Harty et al. (2016), The Mobile Analytics Playbook

¶¶https://www.appbrain.com/stats/libraries/details/sentry/sentry
Overview of the case studies

▶ Azetone: the second of the two ‘heatmapping’ tools. Their online presence indicates they continue to provide a similar product and service.
▶ Countly: one of the early opensource mobile analytics offerings, they offered a relatively simple complete end-to-end package with the client-side SDK and server.
▶ MixPanel: an early closed-source mobile analytics offering.
▶ Segment: another long-term opensource mobile analytics offering, acquired in 2020 by Twilio. Their source code and issues database include various useful exemplary examples; these have contributed to the grey materials used in this research.

5.18 Summary of the case studies overview

This chapter covered a lot of ground. In summary the research includes learning how mobile analytics tools are used from the developers’ perspective across a range of mobile app projects. It also presents some action research, software experiments, and analysis of Android apps that use the most popular mobile analytics service: Firebase Analytics. Finally, it introduced tool centric case studies and finally miscellaneous additional sources of research material.

The next three chapters are a set and structured around the six perspectives. They provide findings aligned with three distinct focal points: first the processes used by app developers; second the products of their work, particularly in terms of improving the product artefacts. The third chapter in the set focuses on the mobile analytics tools and related services. Each chapter considers the status-quo and improvements to the status-quo. Sometimes they build on work presented in another of this set of chapters, for example they may cross-reference each other in their respective discussion sections.
Findings
The learner always begins by finding fault, but the scholar sees the positive merit in everything.

Georg Wilhelm Friedrich Hegel - 1770-1831

This chapter presents the research findings based on two of the six perspectives: uUse and iUse, i.e., how app developers currently use mobile analytics and ways the use could be improved.

The analysis of the data collected from the different case studies identified 41 individual ‘L1’ themes, which are listed in the thematic analysis spreadsheet. Examples of the spreadsheet and of the data analysis performed to derive these themes can be found in Appendix A. The L1 themes were grouped into 4 L2 themes; they were also ranked by frequency, to help select the most relevant examples for this chapter. Figure 6.1 shows the mapping and key relationships between the themes.

The four level 2 themes are covered in this chapter: 1) motivation, 2) Situation Normal All F**ked Up (SNAFU), 3) competence & optimisations, and 4) benefits & achievements.

Motivation investigates what drives the developers to use mobile analytics to improve their apps, for instance is it ‘fire-fighting’ or ‘optimisation’? Section 6.1. Motivating factors presents evidence for this overall theme, and the topic is discussed in the Discussion section of this chapter.

SNAFU: the context the developers are working in plays a major factor in what happens in terms of their use of mobile analytics; the evidence for this theme is covered in Section 6.2. SNAFU and real-world conditions.

Competence and optimisation are vital factors in developers being able to obtain the benefits and gain the achievements of using mobile analytics to improve their apps. Section 6.3. Competence and optimisations presents evidence for this overall theme.

Benefits and achievements of using mobile analytics to improve apps are discussed next, to understand what benefits developers can accrue and what they can achieve. Section 6.4. Benefits and achievements presents evidence for this overall theme.

The chapter also describes how interventions undertaken during the research increased the use of analytics by development teams (Section 6.6. Improvements to the use of Mobile Analytics), before discussing the findings relating to improvements to the use of mobile analytics (Section 6.6).
Findings: Analytics in Use to improve mobile apps

6.1 Motivating factors

Motivation to use analytics spans a continuum from tactical behaviors to avoid and restrict failures, to strategic behaviours aimed at improving the design of the app and engineering practices of the team more generally. This section covers these motivating factors, identified as individual L1 themes, in turn. Note: some examples extend beyond a single factor, and some topics are grouped under a common heading, i.e., the Personally Identifiable Information (PII) topic will be covered in Section 6.1.4.

6.1.1 Successful apps

Developers and their organisations want their mobile apps to be successful. How success is determined can vary from one individual developer to another and from one organisation to another. Nonetheless, unreliable apps are less likely to be considered a success by the developers or their organisation, which aligns well with the findings in this research.

Improving app design: Another factor motivating use of mobile analytics is to improve the design of the app in terms of the user experience (UX). A more reliable and stable app is likely to improve the experience of users; this was a topic mentioned by several projects including Moonpig, Moodspace, and SmartNavi. It’s not always easy to get the balance...
right in terms of informing users that the app contains mobile analytics – informing users may paradoxically reduce the user experience, as illustrated by the Pocket Code app in Section 7.3.4. User Experience of apps with in-app analytics. At the time of the Catrobat case study, the Android app had pages of mandatory text presented to new users of the app.

6.1.2 Motivating devs to use mobile analytics

As the CTO of LocalHalo stated: “If you have lots of crashes you have zero chance of being promoted [by the app store].”

The use of additional in-app analytics is not confined to the commercial apps; the Greentech Apps Foundation (GTAF) team explained how they had actively used mobile analytics to reduce the crash rate of their key apps and announced their intention to increase their use of mobile analytics in order to improve the user experience of their apps [218].

An interesting example from grey literature is the use of an error budget, where the developers have 20% of their time available to work on technical improvements [219]. If the app’s crash-free rate is above 99.9%, the team can choose their work freely; however, when the crash rate exceeds 0.1%, then the developers must spend their 20% fixing the problem (described as the situation in the literature). In this example, developers are motivated not only to use mobile analytics, but also to fix the issues mobile analytics finds.

**Improving crash rate:** this motivation featured in the work of many of the app-centric case studies. For example, the GTAF project reported that several of their Android apps had experienced high crash rates, which had adversely affected the user experience. The project team have learned the importance of paying attention to crashes being reported by mobile analytics tools. Kiwix and Catrobat project leads also reported a desire to reduce the crash rates; in both these cases, the crash rate for their key apps far exceeded the Bad Behavior Threshold of 1.09%, specified by Google Play [220]. Note: a worked example of the Kiwix hackathon experiment which was one aspect for the Kiwix app-centric case study is in Appendix A, in Section A.1. Kiwix experiment (at hackathon).

**Excessively high crash rates motivated action:** In terms of the app-centric case studies, only one of the interview-based studies – GTAF – explained explicitly that they had periods of fixing high crash rates. Several of their Android apps had experienced high crash rates, which had adversely affected the user experience. The project team have learned the importance of paying attention to crashes being reported by mobile analytics tools. They also realised the importance of addressing high crash rates. Accordingly, the development team used outputs from mobile analytics to identify the most frequent crashes they wanted to fix. In contrast, the majority of the rest of the interview-based studies were working predominantly to maintain good quality, to varying degrees, based on their active and ongoing use of mobile analytics.
All of the action-research-based app centric case studies were prompted by excessively high crash rates, i.e., they acknowledged that they wanted/needed help both to tame the high crash rates and to improve the rates further on an ongoing basis.

For Kiwix, there was a pernicious, insidious increase in the crash rates for several of the most popular Android apps. The project leads particularly wanted to reduce the crash rates for the worst offenders in terms of the apps. Similarly, the crash rate for Pocket Code far exceeded the Bad Behavior Threshold of 1.09% [220], and, despite the various recommended software development practices, the project team had not been able to tame the excessive crash rate.

CI’s Android app had a user-base of millions; however, bouts of issues in several releases of the Android app meant that the app had exceeded the Bad Behavior Threshold of 1.09% significantly. They needed to solve the tactical issues in parallel with improving the engineering practices to reduce the likelihood of similar bouts of issues and to encourage ongoing and long-lasting improvements to the quality of their app and the service.

6.1.3 Choosing mobile analytics

The commercial app-centric case studies all incorporated in-app analytics into their apps in addition to error reporting. Firebase Analytics was the most popular choice, which is congruent with Firebase Analytics being the dominant market leader [221]. LocalHalo chose Sentry, in part as it integrated well with React Native, the cross-platform development framework they used for their mobile app. The CI project chose to include several mobile analytics SDKs in the app; Microsoft App Center was incorporated for crash and error reporting.

As mentioned previously, several projects including LocalHalo chose to use a separate mobile analytics service – Mixpanel – for business analytics.

The choice of which mobile analytics tools to use is in some ways actually a recursive choice. First, there is the choice of whether to include any analytics within the app at all. The Kiwix project team chose not to in order to maximise the protection of end users’ privacy and minimise the potential for repressive authorities finding and then abusing usage-related data to penalise end users of Wikipedia and similar material. The rest of the app-centric case studies included at least one in-app analytics SDK in their app.

Another choice is how many mobile analytics SDKs to incorporate into the app. Some of the projects, including LocalHalo, GTAF, and the Commercial project, chose to incorporate several mobile analytics SDKs in their apps. Their reasons differed; for example LocalHalo used Mixpanel for business analytics and Sentry for technology-related analytics including crash reporting. From the interview correspondence with their lead developer, Greentech apps (GTAF) appeared to have let the developer(s) of each app decide on which mobile analytics service to incorporate. This was corroborated by static analysis of their Android apps using the online Exodus Privacy project. For the CI corporate project, the reasons...
were multi-factored and confidential; therefore the factors are outside the scope of this dissertation.

Choices of SDK may change over time. The first choice is whether to continue using an SDK at all. For example, the Catrobat project chose to remove in-app analytics from their flagship Pocket Code app as the replacement for the Fabric Crashlytics SDK – Firebase Crashlytics also collected and combined PII data with the crash reporting. Like the Kiwix project team, the Catrobat team wanted to err on protecting the privacy of their users (which includes many school-age children) and reduce their contributions to the already-vast digital footprint owned by global tech superpowers.

Second, there is the choice of which source to use for which purposes. For instance both Android Vitals and the many crash reporting SDKs report on app crashes. All the projects who used in-app analytics for crash reporting preferred to use that service to monitor the crashes in their app, even if some crashes aren’t reported in that tool, for example those that occur as the app is starting up. Of the development teams who used in-app crash reporting, Moonpig were unusual, as they also actively and frequently used Android Vitals to monitor crashes and other failures.

Third, there are choices to be made about what to record and when to do so in terms of in-app mobile analytics. Various in-app mobile analytics SDKs provide facilities for error reporting in addition to crash reporting. Broadly, errors in this context are connected to caught and handled Java exceptions. Some SDKs include API calls to provide trails of ‘Breadcrumbs’ that lead up to an error or a crash. None of the app-centric case studies mentioned actively using these breadcrumbs; however, the reports Sentry generated for the LocalHalo app indicate that the Sentry SDK generated automatic Breadcrumbs, and these helped identify that LocalHalo’s server was no longer functioning correctly in late 2021.

Caught exceptions, signals, etc. are not the only form of error that might be of interest to developers. Projects including SmartNavi and Moonpig used in-app analytics for reporting additional errors, for instance errors that affected the user experience. Similarly, the in-app analytics and the crash reporting SDKs both collect contextual and runtime information, and developers sometimes use this information.

Too much effort: Paradoxically, sometimes developers choose not to apply the results of mobile analytics. Developers may perceive that the effort required for something is too much, and may therefore take alternative action (including inaction). Examples from the case studies include developers choosing to ignore Android Vitals. Sometimes they do this as they’ve chosen to use another mobile analytics service that provides some of the information Android Vitals provides, particularly on crashes. An example is from Moodspace, where the CTO noted: “ANR & crashes: I usually use crashlytics, so never really use this tab. The reason being, that the play console used to be very unreliable for crashes.” Others, e.g., GTAF, chose to ignore ‘hard to fix’ issues, a topic discussed later in this chapter.
**Developer-controlled characteristics:** Developers have a choice of how much to use an in-app analytics SDK, and similarly they can choose whether to have specific configurations for different environments, e.g., for pre-production environments vs. production. As illustrated in [9], 50 of 107 projects only used the minimal default initialisation, rather than calling the rest of the Firebase Analytics API. None of the app-centric case studies relied on the minimal default initialisation; they were motivated to use the SDK more extensively.

In the commercial case study, C1, pre-production release candidates used a distinct client-ID so that the analytics outputs were separated from production traffic. The app was also configured to generate additional logging in pre-production builds.

### 6.1.4 Ethics and Personally Identifiable Information (PII)

The topic of PII has been mentioned several times already. A related topic is the ethics of collecting mobile analytics data at all, and, if so, whether to default to opt-in or opt-out, and then in terms of what data gets collected, by whom, and for what purposes.

As mentioned in Section 6.1.3. Choosing mobile analytics, the Kiwix project chose not to incorporate any in-app mobile analytics or data collection for ethical reasons: to protect the privacy and provide safety for end users who might be penalised heavily for even using the Kiwix app and possibly imprisoned, or worse. In contrast, the Catrobat project was initially keen to extend the use of in-app analytics, until they realised the type and amount of data collected by the mobile analytics SDK they had used.

#### Comparisons with the ‘web’

In the related domain of websites and web apps, similarly, there are challenges and legislation that pertains to the ethics of the collection of both cookie-related and other data.

The design and implementation of mobile apps has led to an app packaging any tracking or analytics within the Application binary and although there are parallels with first and third party tracking, apps don’t make that distinction for end-users, nor do they appear to have the need to ask users to accept ‘cookies’ as the app doesn’t actually use exactly the same mechanism for tracking.

In practice mobile apps appear to be ‘behind the curve’ compared to web apps in terms of user consent. It might be at par in terms of how Google combines the analytics data from multiple apps using Firebase and/or Google Analytics embedded in apps, an assessment of changes Google made to web-based data collection is described in [222].

Several of the app development teams had to make explicit decisions whether to use mobile analytics and, if so, which one they would use. As mentioned earlier, the Kiwix project team chose explicitly to exclude mobile analytics from their apps to reduce the potential for end users to...
be tracked and potentially imprisoned through their use of the app and Wikipedia content in particular.

The Kiwix team did decide to use the platform-provided analytics from Google Play Console on the grounds that if the end-user had a) installed the Kiwix app from Google Play and b) had enabled (or at least not disabled) Google’s collection of usage and performance data from their phones, then the users were not being compromised by the devs using the analytics generated from the data those phones and tablets had provided.

The Catrobat team started by adding Fabric Crashlytics into their Pocket Code Android app several years before becoming a case study for this research. After seeing the results of applying the techniques from this research, they chose to add Crashlytics to their iOS Pocket Code app and were also planning to use Firebase Analytics to record more granular usage data to both the Android and iOS apps.

However, as a side-effect of migrating from the Fabric Analysis to Firebase Analysis – using the same client side SDK release – they discovered that they started receiving demographic data in tandem with the reliability analytics. Google then set a deadline by which developers would need to replace the Fabric client side SDK with a similar one from Firebase to continue receiving any of the reports. The Catrobat team decided to stop using Crashlytics entirely, rather than having demographics meta-data collected by these SDKs in their apps. The Catrobat team had no objections to continuing to use the platform-provided analytics.

**Impact of ethical decision to remove mobile analytics:** The Catrobat project team chose explicitly to remove support for Crashlytics in the Pocket Code app when Google’s deadline for migrating to Firebase Crashlytics expired. At the time, the project team discussed whether to keep Google Analytics support in the Pocket Code app. Pull Request 3832 in the Pocket Code repository provides a written record of the changes made to the codebase and the discussion.

6.2 SNAFU and real-world conditions

Software development, including developing apps, takes place in the real-world, where there are often multiple conflicting demands on the time of participants – the developers. Therefore, absolutes, superlatives, and other purist approaches are unlikely to work. Bugs will happen; automated tests, if written at all, won’t necessarily test much or test in depth; bugs will happen; nothing will be perfect.

The impracticality of perfection leads to real-world consequences, to coping strategies, to pragmatic behaviours, and so on. There’s a term, SNAFU, which aptly and succinctly describes their working context. Several examples of SNAFU emerged in terms of a) developing apps (theme: poor reliability by default), b) in the development teams using mobile analytics (theme: friction, opacity, inability to act), and c) weaknesses in dev practices, where an accidental misconfiguration pre-release that led to a mute release of Pocket Code that caused the analytics data to disappear.
**Poor reliability:** First, in terms of developing apps, there is poor reliability of the apps in use by default. Kiwix, Catrobat, GTAF, and commercial project C1, all had ongoing periods when the reliability was excessively poor and far beyond one or more of Google’s Bad Behavior Thresholds [223]. Indications from the evidence gathered during this research is that poor reliability accretes when developers do not actively monitor mobile analytics and do not address the more severe of the failures that are reported. Conversely, when developers do address these failures, the reliability of the subsequent release of the app generally increases – a topic discussed in the next chapter in Sections 7.3.1 and 7.3.2.

**Friction, opacity, inability to act:** The second SNAFU factor is when developers do not have access to mobile analytics outputs, or when they do have access but are not able to use the outputs productively. Surprisingly, this was commonplace in the app-centric case studies. As access to the mobile analytics tools is restricted by default⁷, someone has to grant access to individual developers. For LocalHalo, only the CTO and the researcher had access to Sentry; the other developers at LocalHalo did not have access. It is not clear why the rest of the development team lacked access.

Especially for larger teams, and teams where participation is periodic, many team members are not granted access. Three of the app-centric case studies exemplify this: Catrobat, Kiwix, and the commercial project C1. There were two exceptions where all the developers had access: Moodspace, where the sole developer had access, as he was also the account owner; and Moonpig, where all the developers were provided with at least read access to Google Play Console with Android Vitals and to Firebase Analytics.

The lack of access for team members increases the importance and value of populating bug reports with pertinent information from the mobile analytics report(s).

**Weaknesses in dev practices:** The third example of SNAFU is for the Pocket Code project. An accidental misconfiguration during the release process led to the loss of Fabric Crashlytics data for that release. Fortunately, Google Play Console with Android Vitals continued to provide ongoing outputs indicating the benefits of having platform-level mobile analytics. The development team applied the correct configuration for the following release of the app, and Fabric Crashlytics data was restored for subsequent releases.

The worked example from LocalHalo in Appendix A in Section A.2 provides an example of weaknesses in the dev practices; the CTO had access to both Sentry and Android Vitals but did not happen to look at them when there was a spike of high crash rates over several days. Perhaps the issues would have been found sooner if they had provided access to both these mobile analytics services to the rest of the development team.
6.2 SNAFU and real-world conditions

6.2.1 Engineering tradeoffs

Software developers have to deal with tradeoffs, for example between developing features versus improving the existing codebase. In terms of the stability and reliability of their apps, there were various observed examples where engineering tradeoffs were made. For example, the Kiwix project chose to substitute their very capable download manager for the basic functionality provided as part of Android.

Another tradeoff made by the Kiwix team was based on the perceived maintenance overhead of making interim releases of the app; as a result, the development lead rejected the proposal to make interim releases for various of the custom apps, even though an interim release was likely to improve the reliability of those apps significantly. A secondary consideration was the burden of creating and rolling out releases of the custom apps; at the time, the release process was burdensome for the developers.

Moonpig took several months to first replace the third-party RoboSpice SDK and then deploy the new release without that library. Release management is a key factor in decision making. Crash analytics helped quantify the effects on groups of affected users (those on newer releases of Android).

Flo Health have explicit tradeoffs between how developers spend their time developing their app. They only spend 80% of the time on core development; the remaining 20% is ring-fenced for other development work. This 20% is either spent on whatever the developers choose to work on, or on addressing errors if any error budget has been exceeded.

Third-party software challenges: Third-party software is not modified directly by the app’s development team (there may be ways they can modify it in other ways; however those are beyond the scope of this research). Therefore, there may be challenges when the app developers want or need changes to the third-party software. Furthermore, the third party may control what modifications are made to that software. Sometimes the app developers lack the context, wherewithal, permission, and so on to effect timely changes to the third-party software, i.e., there may be third-party software challenges.

Using third-party software can lead to additional SNAFUs: A couple of examples from the Kiwix case study illustrate some of the challenges, in terms of using mobile analytics, that app developers need to consider with third-party software that’s virtually part of Android: the closely-coupled Google Chrome browser and the WebView component that provides many apps with an embedded web browser and has over 5 billion downloads. Crashes have been reported frequently with both of these components for the Kiwix apps, amongst many others. For some of the crashes, the developers are able to change the source code of the app to address some of the crashes; however, many seem beyond the immediate control of the app developers – and yet the crashes are counted by Android Vitals and Google Play Console toward the cumulative ‘bad behavior threshold’.

8: They subsequently replaced the Android-based code with an opensource download manager and, in turn, that is overdue replacement at the time of writing.

9: Details in github.com/kiwix/kiwix-android/issues/1426

10: At the time of writing RoboSpice is still in 500+ apps which may be experiencing an ever increasing crash rate as newer versions of Android become more prevalent, source www.appbrain.com/stats/libraries/details/robospice.

[219]: Koutun (2021), How to deal with tech debt at the scale of super app

[224]: Google Inc. (2022), Android System WebView

11: For example even Google’s GMail app, Facebook’s app, and the BBC apps were adversely affected in March 2021 [14].
The commercial project C1, like many, used the third-party opensource OkHttp library, which is in at least 6% of top apps and 5% of all apps measured by AppBrain [225]. To the OkHttp project’s credit, they provide various automated tests which can help both developers of OkHttp and users (i.e., developers) of software that incorporates OkHttp. As the feature set of OkHttp increases, so can the complexity of using these various features – as the commercial project discovered when a developer inadvertently made changes to improve logging that led to the crash rate increasing several fold. A key challenge for the development team was first finding a fix and then testing that the fix was effective before the revised app was released in the app store (rather than waiting for mobile analytics to report any change in the measured failure rate). They did so with the benefit of building on parts of the existing opensource OkHttp codebase.

Grey data sources are frequently used by app developers to discuss problems with third-party components, with the two main homes for these discussions being GitHub projects and StackOverflow.com.

**Release Management**: Chapter 2 in Section 2.1.5 introduced three phases of a release. Release management encompasses all three of them and release management reports are most relevant for the last of these, the deployment phase.

Two examples of release management have already been mentioned (for Kiwix custom apps and for Moonpig, both in Section 6.2.1. Engineering tradeoffs); the C1 project also placed great importance on scheduling releases with the intention of maintaining goodwill and encouraging adoption of the new release.

Google Play Console incorporates various aspects of its reports, including various pre-filtered reports from Android Vitals into a Release Management section of the online Graphical User Interface (GUI). The Release Management section includes mechanisms to perform a staged rollout. Staged rollouts are particularly pertinent for apps with large userbases, as they help the development team a) obtain focused information about the reliability and the adoption of the new release, while b) also constraining the risk of a release failing in production [12].

The product owners for the C1 project learned about how to combine the Release Management reports (including Android Vitals aspects in particular) and Microsoft App Center’s reports, in order to have greater insights into the vital signs and overall health of new releases. Android Vitals was seen to provide fast feedback into the health of a series of releases of the Android app, and it was a highly effective early warning indicator of reliability issues emerging in a new release. Of note, staged rollouts cannot be reduced, nor can staged rollouts be targeted to users of a particular release of the app, so it’s not currently practical to perform incisive upgrades to replace a failing release without also upgrading some portion of devices that serve the other users of the app.

**6.2.2 Sense-making and decision-taking by developers**

Beacon-finding and drill-down parallel similar practices used by app developers when they use mobile analytics as inputs to their development
work and as feedback for [their] previous development work. One example is from Moonpig where they identified the pattern connecting the crash rates and Android versions related to the RoboSpice library that was being used in the Moonpig Android app. The RoboSpice GitHub project confirmed the factors that led to the increase in the crash rate were caused by Android tightening the restrictions on background processing in order to save battery life.

There are parallels between the sense-making process used by developers and those performed by end-users, described in [226, pp. 5:15-5:16], where developers forage for information in locations that are likely to contain answers (e.g., on StackOverflow and on GitHub).

Figure 6.2 illustrates the sense-making and triage process used by development teams which shares various similarities with sense-making from a research perspective. These similarities mean the researcher and the practitioner may also share similar practices in terms of their analysis of phenomena found in mobile analytics tools. The triage and drill-down may be repeated several times, when there is sufficient potential value in performing further investigation.

Developers have finite time and resources to address issues reported by mobile analytics; how finite depended on their project and circumstances. The finite nature means they are unlikely to process every issue, so at some point they will cease processing the remaining issues – this is the stopping point in the figure.

The impact of reported failures is combined with situational risk-assessment as a part of the decision-making process performed by developers during...
6.2.3 Sources of suboptimal practices

A cluster of several related themes – weaknesses in development practices, premature satisfaction, what developers notice, and flaws in the mobile analytics – may all lead to sub-optimal practices.

Weaknesses in development practices: Weaknesses in development practices result in failure to address reliability issues. And where practices depend on one individual, they can fail when the bus factor [227] strikes.

Premature satisfaction: An inference based on observation is that many of the developers in the app-centric case studies were often satisficed with what the mobile analytics tools reported – where they accepted local optima (determined through a combination of observation and asking the app devs), e.g., they accepted the ‘top’ crash cluster as the worst one. Therefore, if there are flaws in what is being reported, the effects of those flaws may influence what the developers do and don’t do. Indeed, as shown in Section 8.4.1. Flaws in the mobile analytics tools and/or services, there are flaws in the groupings of similar failures in at least three of the mobile analytics crash reporting outputs.

What developers notice: one of the success factors in terms of optimal practices is what the developers notice in mobile analytics reports, as what they don’t notice does not get acted on, at least not deliberately!

6.3 Competence and optimisations

At a high-level, there are two key competencies: to enable the app to use any of the in-app analytics (details and examples of how to do so are in Section 7.1. Code-facing topics), and then to actually use the reports and
apply the results. The rest of this section presents the other competencies identified in this research.

6.3.1 Using issue databases and preserving information

The GTAF team created issues in their online issue database gitlab.com/greentech/ for crashes, and included links to the source information in the respective mobile analytics tool. These links are a) only available to people who already have access to the mobile analytics account, and b) ephemeral. There are several ways development teams can extend the useful life of the private and ephemeral contents.

In the three action-research app-centric case studies (C1, Catrobat, Kiwix) screenshots of the contents of the reports were combined with extracts of various pertinent statistics, for instance the frequency of a particular crash cluster. These contents were added to issues in the issues database. The developers appreciated the additional information (together with an online link to the relevant report which helped them check for any updates while that report was still available); they said it helped them appreciate the magnitude of the problem, which, in turn, helped motivate them to address the issue sooner. The data was generally collected manually and pasted manually when the issues were raised, in part as automating the preservation of the reports is beyond the remit of individual team members.

Integrating mobile analytics into the practices of the team: One of the app-centric case studies, Moonpig, demonstrated that they actively guarded against entropy through frequent, ongoing checking of the various mobile analytics services they used. They had an assigned developer-on-duty who was expected to check these services during the working day and action any issues or anomalies they noticed.

6.3.2 Engineering leadership

There were clear correlations between having engineering leadership participating and/or supporting the use of mobile analytics, and the measured reliability of the respective apps. Moonpig had integrated the use of mobile analytics in both development and operational aspects of their business. Kiwix achieved manifold improvements and sustained them while the Android project had a professional app developer as the project lead.

In contrast, the improvements in Pocket Code petered out after the loss of the Product Owner, a PhD student. This was corroborated by the project leader, who confirmed that the collapse in improvements was the loss of this Product Owner who successfully completed his PhD and moved to a role in industry. (Here are two useful references on the project team’s approach to software quality [228] and to product owners [229].)
6.3.3 Triage

Triage is where the developers decide what to do, if anything, with failure clusters being reported by mobile analytics. Several of the projects confirmed that they used mobile analytics as a source when the issues were triaged, it was not known whether the rest did so as the topic was not discussed explicitly with every development team.

Broadly, the triage process results in one of four outcomes:

1. Accepted issues: which the project team intend to address.
2. Pending issues: where the project team need more information before being able to reach a triage decision.
3. Explicitly rejected issues: which the developers chose deliberately not to action, for examples issues considered impractical to address. Note: Some mobile analytics tools provide a facility for developers to ignore or mute these issues.
4. Implicitly rejected issues: for some projects this applies to the majority of the failures reported by the various mobile analytics tools.

The triage process sometimes led to issues being raised in the respective issues database for at least some of the issues they chose to action. For example, Catrobat reported them in Jira; Greentech Apps (GTAF) used GitLab’s integrated issues database; and Kiwix similarly used GitHub issues. Moonpig often raised issues; however, sometimes instead they simply modified the code where they believed the fault was clear and the fix was quick and easy to do. The C1’s development team used recorded issues in their organisation-wide issues database. However, projects did not necessarily record every issue they actioned.

6.3.4 Hard to fix issues

The GTAF team used a heuristic when deciding which crashes to fix. They chose to work on the issues they perceived as relatively easy to fix and which affected many users. They provided examples of easier-to-fix exceptions (NullPointerException\(^{13}\) and IndexOutOfBoundsException\(^{14}\)) in contrast to some they found harder to fix (IllegalStateException\(^{15}\) and native crashes\(^{16}\)).

SmartNavi found that it was not practical to identify or try to resolve several low-frequency crashes that were only reported for unfamiliar and unavailable devices in China. As there was a sole developer on a voluntary project, the developer had little scope to invest significant time and resources.

6.3.5 The fix process

Fix processes for issues found from mobile analytics are similar to fixing issues found from other sources, for instance: from testers, from app store reviews, and so on. There are some particular nuances when mobile analytics is the source; for example, the mobile analytics report will generally include one or more instances of a stack trace, together
with precise meta information. Mobile analytics also includes analytics such as any patterns of the failure\textsuperscript{17}.

The combination of failure details, meta information, and patterns often helped development teams prioritise, manage, and diagnose bugs reported in mobile analytics more effectively compared to bugs from other sources. Also, mobile analytics was able to provide measurements of any improvements (and any degradations) of the updated app.

The teams varied in their how they fixed issues. Some fixed the source code directly without externalising the reproduction or testing aspects; others attempted to externalise one or more of bug identification, bug reproduction, and testing practices. Those who were working tactically seldom invested time to reproduce the failure or write automated tests; instead they relied on external sources to provide feedback, e.g., from testers and/or from mobile analytics reports for the new release.

For the app in the C1 case study, two of the developers invested the time to distill the cause of the crash to a minimum, reproducible example\textsuperscript{230}. This was written in pure Kotlin. By using Kotlin, they could run directly in the JVM, so the Android runtime was not needed. The Android runtime complicates the runtime conditions, and the tests run much more slowly as tests that need the Android runtime. A set of automated tests was created to reproduce the crash at will. The flawed code was then improved so it no longer crashed and performed all the intended actions (previously it crashed instead). The automated tests demonstrated that the fix was complete and correct. All the source code was then integrated into the app’s codebase. (The tests continued to run directly in a JVM rather than needing an Android runtime, so they could run quickly. Doing so is a mature development practice by Android developers and not unique to this app or development team.)

The fix process therefore generally includes: a) an assessment of any recognised patterns in the failure to help with bug management, and b) post-release monitoring of mobile analytics to see if the fix actually improved the measured reliability of the app in the new release that includes that fix. While it is possible for mobile apps to include in-app dynamic updates of code, these are a) uncommon and b) discouraged by the app stores. Note: Google Android has an API called in-app updates\textsuperscript{18}; however, it is more of a mechanism provided to enable app developers to write code that can query the app store for newer releases and for whether the user should be asked, encouraged, or required to perform an update. In-app patches of an app binary are therefore outside the scope of this research.

\textbf{Fixing crashes may be necessary but not sufficient:} One of the issues\textsuperscript{19} and associated bug fixes\textsuperscript{20} pertaining to the Pocket Code app and the hackathon\textsuperscript{21} has a discussion written by one of the project leads in the Pull Request\textsuperscript{22}. He observed there are still issues where content is not downloaded; however the app no longer crashes. The high crash rate was sufficient to motivate the developers to address the source of the crash in the source code. However, more work was needed to fix the failures to download and apply content. In the Pull Request, the developers also discussed the ramifications of removing the visual indicators when the user triggers a download and decided to simply try to fix the crash. As

\textsuperscript{17} As an aside, several development environments can directly process and help analyse a stack trace for an app, for example in IntelliJ (which also underpins Android Studio used by many Android developers) jetbrains.com/help/idea/analyzing-external-stacktraces.html.

\textsuperscript{230} Stack Overflow online help (2022), How to create a Minimal, Reproducible Example

\textsuperscript{18} developer.android.com/guide/playcore/in-app-updates

\textsuperscript{19} jira.catrob.at/browse/CATROID-379

\textsuperscript{20} github.com/Catrobat/Catroid/pull/3362

\textsuperscript{21} Via jira.catrob.at/browse/CATROID-405

\textsuperscript{22} github.com/Catrobat/Catroid/pull/3362#issuecomment-541055675
the developers could not reproduce the crash, they were not confident in their ‘fix’ working.

6.3.6 Integration of mobile analytics

Integration of the tools and systems in this context facilitates and improves the ‘working together’ of mobile analytics with other software tools as performed by the app developers. It ranges from manual and often short-term mechanisms, to streamlined pipelining of mobile analytics outputs into other systems. Often there’s a basic facility provided which enables issues to be raised directly from a mobile analytics web page.

The majority of the app-centric case studies partly integrated mobile analytics reports into their bug tracking process by storing at least some of the contents of the reports some of the time.

Of the projects that recorded the failures found by mobile analytics, it appears the Catrobat project used the crash’s stack trace without anything else of the crash cluster’s details or analytics. Moonpig stated that they recorded many of the failures; however, sometimes they short-circuited the bug tracking system and modified the source code directly, when the likely fix was believed to be easy to address with minimal risk. As they monitored the various mobile analytics services assiduously post-release, they had at least one safety net.

Kiwix, Catrobat, and LocalHalo only provided a subset of the development team access to the mobile analytics services. As a result, the rest of the team were blind to any reports or issues unless someone who has access provides the relevant information in the bug tracking system. The GTAF project team record crashes reported by Firebase in their bug tracking system; however, they only embed the URL, and they had not included copies of the pertinent details.

As can be observed in the Greentech project’s issues database, for example, the developers include the URL that points to the pertinent report in the source mobile analytics service. These links work for a period, typically hours to weeks, and are limited to people who have access to both systems. They enable authorised users to check the source material from the relevant issue quickly.

These links eventually decay once the source content is no longer available, as discussed in Section 8.2.3. Integration into workflows. In the three main action research projects, screenshots together with pertinent text-based extracts were included in the issues database to help preserve the information indefinitely. Similarly, Vitals Scraper was developed and used to automate the collection of screenshots and the underlying data from Android Vitals, in order to preserve the information and make it available for additional analysis.

Microsoft App Center, for example, has a facility which enables comments to be added to failure clusters. In the C1 project, developers sometimes added a key in the comment field to indicate that this failure cluster was under investigation. The key was often a bug reference, and the developers sometimes also included terse notes. Sadly, it was impractical to determine the criteria for the various behaviours by the developers, as the team were
working remotely on another continent, and the commenting mechanism did not show who provided the comment, or when.

Despite best intentions by the app developers on multiple projects there were examples where developers did not add the pertinent cross-referencing between the issues database and the mobile analytics service.

At least one of the projects, C1, used APIs provided by the mobile analytics service to systematically export the contents of the crashes and errors that were captured in that tool: Microsoft App Center. Doing so enabled the data to be mined in combination with various other sources of data; however, the details are beyond the scope of this PhD.

6.4 Benefits and achievements

Development teams were able to benefit from using mobile analytics in various ways, e.g., to improve their engineering practices, to respond and act sooner, and to perform proactive fault finding. By being informed through mobile analytics of the magnitude and scope of various failures, they were able to work more effectively and take better control than when they did not have mobile analytics available to them. They were able to achieve more.

6.4.1 Proactive fault finding

Moonpig used Firebase Crashlytics and Google Analytics for diagnostics. They spent 30% of time spent using Android Vitals to identify flaws and issues related to their Android app. They noted that, while finding and fixing causes of crashes may be quick, the release may take several weeks before it’s deployed.

In the C1 project, the use of mobile analytics helped identify a major issue within days. Automated tests were developed that reproduced the failure, and they helped establish confidence in the tests and in the subsequent fix to the networking code. Proactive use of the Release Management reports in Google Play Console helped the team release the revised app with visibility into the effects of users updating and using the revised app. In general, proactive use of the release management reports helped the team make immediate interventions when there were indications of releases having adverse effects.

Note: In addition, training was provided to the Product Managers for the C1 project on how they could use Google Play Console, Android Vitals, and the Release Management reports to provide them with direct visibility into the state of the app and indications of the effects on the end users, e.g., through the installation and de-installation reports. This was one of the actions that helped scale the team’s capabilities.

The majority of the app-centric case studies were known to use aspects of the Release Management tools in Google Play Console. The rest may also have done so; however, the information is not available from those case studies.

25: Note: several other mobile analytics tools also provide similar API integration points, some of these will be discussed in Section 8.2.3. Integration into workflows.
6.4.2 Bug reproduction

Being able to reproduce bugs, ideally at-will, enables developers to work more effectively at addressing the bug and having confidence that their intended fixes actually do fix the bug. Bug reproduction can increase the developer’s confidence the bug is real and can be managed, i.e., it is under control. Then they can make changes to the the software to determine whether their changes address (fix) the bug.

The developers are not always able to reproduce bugs. For example, the developers of the Pocket Code app had to apply their ‘fix’ blindly with the immediate aim of fixing, i.e., preventing, the crash github.com/Catrobat/Catroid/pull/3362. As mentioned in the previous subsection, in the C1 project, two experienced engineers spent several days devising a consistent and clear way to reproduce a crash in the response handling and logging for proprietary code that interacted with the very popular OkHttp opensource library (more details are in the section on the fix process in Section 6.3.5). This effort was deemed necessary and appropriate given the strategic importance of the app to the business, and the effects of the crash on a significant subset of users’ sessions.

The research also had mixed results reproducing bugs that were found and reported by mobile analytics. These were aimed at reproducing crashes that occurred in the Kiwix app. The Kiwix project included crashes in several software components Google provides android developers: the WebView (an embedded browser), and Chrome.apk - the binary file for the Google Chrome Android app. It also included attempts to reproduce crashes in the app’s codebase.

- One experiment used the same model of Samsung Android smartphone that incurred a higher-than-usual crash rate; however, there was insufficient information available in Android Vitals to determine enough of the events that led to the crash or any of the contextual information. The manual interactive testing did not manage to reproduce the crash.
- Another experiment scripted the semi-autonomous Android Monkey testing utility, where the script seeded each test run to make the sequence of generated inputs consistent with the aim of first triggering crashes and then having sufficient information to reproduce those crashes at will.

With the exception of the C1 project, none of the other commercial projects (Moonpig, Moodspace, LocalHalo) provided examples of being able to reproduce crashes and other failures that were reported by mobile analytics; nonetheless they chose to use in-app analytics and the respective mobile analytics reports in preference to Android Vitals, as they believed these tools provided more relevant reports with the necessary information to enable the developers to find and fix the causes of those crashes.

There are plenty of instances where app developers had to debug and try to fix failures without being able to test the code confidently to a) reproduce the failure, or b) evaluate the changes to the source code. Releasing the new version of the app and monitoring the effects became part of their practice; albeit some projects monitored more assiduously than others. This approach has parallels with the approach described by
6.4 Benefits and achievements

The GTAF project found crashes pre-launch by using – and paying attention to – the pre-launch reports provided by Google Play Console. Many of the case-studies used these pre-launch reports.

For the Kiwix Android app, the pre-launch report for a newer release (App version: 7230406.aab) showed results that indicated a degradation in the performance of the app compared to a previous release (App version: 7230405.aab). And a crash caused by a java.lang.IllegalArgumentException Parameter specified as non-null is null: method org.kiwix.kiwixmobile.core.utils.LanguageUtils.getCurrentLocale, parameter context occurred with a locale of ‘ar’ (Arabic) on a Sony G8441 device, running Android 8.0.

A strength of the pre-launch reports is the automatic cross-referencing between crashes that were discovered during the pre-launch testing and those that occurred in that release subsequently in production. Figure 6.3 provides a good example, in which a Java OutOfMemory (OOM) error was detected and a link to Android Vitals provided directly in the report. At that time, the app had various memory leaks which the developers were identifying and addressing [232], and OOM errors occurred when

Ghardallou et al. [231]; however, they aim to prove relative correctness, whereas using mobile analytics aims to measure characteristics of the software when in real-world use.

6.4.3 Pre-launch testing

The GTAF project found crashes pre-launch by using – and paying attention to – the pre-launch reports provided by Google Play Console. Many of the case-studies used these pre-launch reports.

For the Kiwix Android app, the pre-launch report for a newer release (App version: 7230406.aab) showed results that indicated a degradation in the performance of the app compared to a previous release (App version: 7230405.aab). And a crash caused by a java.lang.IllegalArgumentException Parameter specified as non-null is null: method org.kiwix.kiwixmobile.core.utils.LanguageUtils.getCurrentLocale, parameter context occurred with a locale of ‘ar’ (Arabic) on a Sony G8441 device, running Android 8.0.

A strength of the pre-launch reports is the automatic cross-referencing between crashes that were discovered during the pre-launch testing and those that occurred in that release subsequently in production. Figure 6.3 provides a good example, in which a Java OutOfMemory (OOM) error was detected and a link to Android Vitals provided directly in the report. At that time, the app had various memory leaks which the developers were identifying and addressing [232], and OOM errors occurred when

Ghardallou et al. [231]; however, they aim to prove relative correctness, whereas using mobile analytics aims to measure characteristics of the software when in real-world use.
testing the app on some low-end devices [233]. The devs were using LeakCanary, an opensource tool designed to help Android developers identify memory leaks.

Various developers have reported false positives 26, and this error occurred in several of the case studies including Kiwix and Cl. These false positives sometimes led to developers choosing not to use the pre-launch reports.

6.4.4 Improving engineering practices

Moonpig’s engineering team used mobile analytics strategically and tactically to nip any unreliability issues ‘in the bud’, i.e., quickly and practically, in order to minimise adverse effects on the end users and the business.

They incorporated Firebase Analytics for usage, crash, and error reporting, and they had developers on rotation to monitor and triage any issues as soon as practical – within hours – after they arose in the mobile analytics reports. When unknown situations occurred, the developers increased logging in the app (using Firebase Analytics as the conduit for the logging, similar to the use presented in Section 5.10. Augmenting the app-centric case studies: sourcecode analysis).

Conversely, several of the case studies demonstrated that inattention leads to entropy in terms of using mobile analytics to maintain the stability and reliability of their apps. Kiwix lost the lead developer in early 2021, and first the crash rate and subsequently the ANR rate increased markedly. GTAF similarly would experience bouts of excessively-high crash rates until they focused their attention on addressing them. LocalHalo hadn’t noticed the loss of reports in Sentry nor the increase in Android Vitals, until the issues were pointed out to them via this research. None of the projects was immune from the adverse effects of inattention; however, Moonpig was by far the most effective at finding and addressing the issues quickly, and their periods of inattention were brief.

Pre-Launch Report (PLR) can be incorporated into developers’ practices to improve the quality of Android apps, and several of the case studies, including Kiwix, GTAF and Catrobat, use them. The reports are discussed in the Tools chapter, in Section 8.3.3. Pre-launch reports.

6.5 General topics

This section presents more general findings that relate to analytics in use, that range beyond any single one of the level 2 (L2) themes in this chapter.

6.5.1 Scaling the use of analytics

This topic presents two forms of scaling. The first is scaling-out the use of mobile analytics to more team members. The second is scaling up through the integration of mobile analytics where the data can be analysed at scale and combined with other large-volume data sources.
**Developers’ access to, and use of, mobile analytics services:** In hindsight, it may be blindingly obvious that, in order to use mobile analytics, one first needs access to the mobile analytics service and then one needs to gain the habits and understanding to actually use what’s been made available. And yet, in the majority of the case studies, only a minority of the team leaders actually provided access to the majority of their team members. Examples of these case studies include LocalHalo, where the CTO appeared to be the only developer with access to Sentry; in the Kiwix Android team, where lead developers were the majority of those who had direct access to Android Vitals; and similarly for the Catrobat project, access tended to be for senior developers who were long-time team members.

In the commercial project C1, there were challenges in determining who granted access; subsequently, developers had to obtain prior approval from their manager before requesting access to mobile analytics, and new team members were not necessarily even aware of the mobile analytics services, so couldn’t begin to request access.

The clear counterexample was Moonpig, where the development team members were routinely provided access to the mobile analytics as part of joining the development team. Furthermore, the developers rotated on-duty to actively monitor the mobile analytics to find and respond to any anomalies; therefore, they were encouraged to learn how to use, understand, and act on the reports in the mobile analytics services.

When copies of pertinent information, extracted from mobile analytics reports, are incorporated into an issue tracking system they: a) are preserved generally for further use, and b) help to share the information at the time with the entire development team (assuming they have access, etc.).

**System integration of mobile analytics outputs:** the C1 project was and is part of a much larger engineering organisation with thousands of staff members. The engineering organisation required integration of all the mobile analytics services into their data lake through data-pipelines. Several of the enterprise-scale mobile analytics services, including Google Analytics 360 and Microsoft’s App Center, provide high-volume exports using their respective proprietary cloud storage service. Microsoft App Center also provides an openAPI that can be used for REST queries interactively openapi.appcenter.ms/ and/or programmatically (documented online at docs.microsoft.com/en-us/appcenter/api-docs/). This was used on an ad-hoc basis during the case-study before the data-pipeline was fully established and worked adequately for the ad-hoc work.

Various of the mobile analytics services provided API integration, these are covered in Section 8.6.2. Improving integration.

The Google and Microsoft storage integration requires both authorisation and payment by the organisation. In turn, this meant a development team needs to get approval to access and use these services, and the approval may be time-consuming, even if senior management has dictated that the project will integrate and use these services as a side-effect of corporate processes and working practices. An entirely separate team may be responsible for the ‘plumbing in’ of the project’s data from the mobile
analytics in Use

Figure 6.4: Moonpig: User Review ‘Always crashing in the middle of purchases’

REVIEW FROM DENISE LAVINGTON
Denise Lavington

Always crashing in the middle of purchases. However, it now works well. If it carries on without hiccups it will get 5 stars.

Moonpig.com

Hi Denise, thanks for your review. We’re sorry to hear that you’re experiencing issues with the app. We recommend deleting and reinstalling the app, as we find this helps in most cases. If you are still having issues please call 0345 450 0100 or email help@moonpig.com. Thanks, The Moonpig Team.

6.5.2 Mobile analytics in a larger context

The research focuses on failures identified using mobile analytics and the processes where developers incorporate mobile analytics into those processes. There may be other sources of information about failures, and there may be other causes of fixes and improvements.

Crashes mentioned in user reviews: mobile analytics is one source of information on failures; other sources are also available. For example, in this research, many of the apps have reviews in the app store (Google Play) that mention crashes. The reviews vary in their specificity, and the project teams seldom diagnosed the crash from the description in the review text.

As a concrete example of perhaps the most detailed review, for the Moonpig Android app, on 16th July 2019, Denise Lavington provided a review of the Android app and awarded the app four stars. The review said: “Always crashing in the middle of purchases. However, it now works well. If it carries on without hiccups it will get 5 stars.” Figure 6.4 shows the review in context in Google Play 28.

The developer investigated this reported issue and could not find any crash in Firebase or Android Vitals that might affect purchases. A possible reason for why the crash did not appear in Android Vitals is that the underlying data is only provided by a user’s device if they’ve opted-in to automatically providing their usage and diagnostics data [236] and if the aggregate data exceeds an undocumented threshold (a topic discussed in Chapter 8 in Section 8.2.5).

Monitoring reviews can be useful to identify issues including user dissatisfaction (app reviews are discussed in the Literature, in Section 3.5.2. Reviews in app stores). In this case, there was insufficient evidence to corroborate the user’s report. As a side note, Google Play Console provides automated analysis of reviews, including those that infer stability issues for instance where the review text includes the word ‘crash’ in them [237]. In the author’s experience, the analysis is imperfect; this is a...
6.6 Improvements to the use of Mobile Analytics

Moonpig demonstrated a highly effective use of mobile analytics to detect failures, including Crash clusters, and they also used mobile analytics to gain additional information about possible causes (similar to the concept of using Breadcrumbs; however Moonpig used logging more holistically). The example from C1 of devising automated tests to reproduce crashes that occurred in complex code, and for which the implementation of suitable dynamic tests required non-trivial work, demonstrated that developers who are willing and able to invest the time can gain confidence in the reliability and behaviours of their software under realistic conditions.

As Kiwix demonstrated, it is possible to increase the reliability of a failing app quickly and effectively. The next chapter includes a summary of the improvements (in Section 7.3.1. Improvements in crash rates). In contrast, improvements stalled for the Pocket Code Android app when the work was disrupted for various reasons in 2020.

Some improvements in use depend on the reporting and integration facilities offered by the tools, and similarly on the quality and timeliness of the reports. Aspects of these are covered in Section 4. Empowering developers, Section 9.1.3. Improving analytics capabilities.

Others can be effected using the current services offered by the tools, for example relatively simple things such as enabling the entire team to use the mobile analytics, Section 9.1.1. Scalability, through Section 6.3.2. Engineering leadership that values mobile analytics and encourages using them on an ongoing basis, and by using them regularly, especially before release by using Pre-Launch Report (PLR), and on release by using the Release Management reports.

A pragmatic acceptance that an app will not be truly free of failures, means projects and teams should pick their own targets (these should be significantly higher than meeting the Bad Behavior Threshold). Android Vitals provides quartiles per application category, so a useful heuristic would be to be in the top quartile in terms of app stability.

As SmartNavi explained, some failures are impractical to investigate or address. This was confirmed by the Moonpig case study:

“I don’t know why some crashes don’t even show up but we honestly just accepted that Android is weird on some even weirder devices. As long as our crash-free rate is that high and will go even higher after the next update, we are happy.” (source Moonpig developer)

Interventions to increase use of mobile analytics There were both research-led and project-led interventions to increase the use of mobile analytics.
Research-led interventions: Each of the three app-centric case studies that involved action research included one or more interventions. For the two opensource projects, Kiwix and Catrobat, hackathons were used, as they provided an immediate low-ceremony approach that enabled quick and effective collaboration. For the C1 project, the intervention was performed over a period of approximately six months and involved working with several members of the project’s development team to help them understand how mobile analytics provides them with pertinent information and how to then apply that information to fix the underlying issues.

Project-led interventions: For the Catrobat case study, the project leadership saw sufficient benefits from using mobile analytics after the hackathon that they decided to also implement it in their iOS app. However, several months later they reversed this decision, because of the data collected implicitly by Firebase Crashlytics. This topic is covered in Section 6.1.4. Ethics and Personally Identifiable Information (PII).

6.7 Chapter summary

This chapter presented ways in which mobile app developers used mobile analytics, together with an overview of ways the use by app development teams could be improved. These findings are summarised below and discussed together with the other findings in Section 9.1. Discussion.

- **Motivating Factors:** App developers are motivated to create successful apps; unreliable apps are seldom successful. High reported crash rates motivated many of the development teams to act to address the worst offenders. Developers made conscious choices how many in-app analytics SDKs to incorporate, ranging from zero (Kiwix, Catrobat) to several C1. Some developers prioritised ethical issues in terms of user privacy (Catrobat) and safety (Kiwix) over using in-app mobile analytics.

- **SNAFU and real-world conditions:** Poor reliability accretes when developers do not actively monitor mobile analytics and do not address the more severe of the failures that are reported. Conversely, when developers do address these failures, the reliability of the subsequent release of the app generally increases. There are various engineering tradeoffs app developers consider. Suboptimal practices in terms of not addressing all the reliability issues are common except in high-performing dev teams (e.g., Moonpig and Moodspace).

- **Competence and optimisations:** Effective engineering leadership was a key success factor in engendering competent and systematic use of mobile analytics to improve reliability. When mobile analytics was integrated into development practices, including recording details in an issues database, the larger team was more effective. Details in failure reports helped the developers diagnose and triage issues, and they were sometimes able to fix the issues directly even without reproducing the failure.

- **Benefits and achievements:** Mobile analytics enabled proactive fault-finding, and pre-launch test reports from Google Play Console
were able to identify some crashes before the app was released to end users. Some failures were hard to reproduce, and teams seldom bothered to try to do so; a notable exception was the reproduction of the OkHttp crashes by project C1. Moonpig exemplified good engineering disciplines by updating the use of mobile analytics in the app to collect data quickly and effectively for potential issues.

- **General topics:** These included factors in scaling the use of mobile analytics, and comparing mobile analytics with the contents of reviews in the app store.

- **Improvements:** Research-led and project-led interventions enabled teams to improve their use of mobile analytics. Development teams who are proactive are able to obtain additional information quickly a) by collecting pertinent data using mobile analytics, and b) by devising automated tests that reproduce the failures, so potential bug fixes can be evaluated pre-launch.

The features of analytics tools will be explored in Chapter 8. Findings: Mobile Analytics Tools and their Artefacts. Before this, Chapter 7 presents the effects of mobile analytics on apps and their artefacts.
This chapter presents the findings for the research perspectives associated with apps and their artefacts, i.e. uArtefacts and iArtefacts. The artefacts associated with a mobile app development project are a product (or outcome) of developers’ activity as they create and maintain the app. Beyond the obvious artefacts such as the source code of the app, artefacts include items stored in the bug-tracking or issue management system of the project, as well as outputs from build processes, such as test results, the release binaries and outputs from mobile analytics.

Broadly, as shown in Figure 7.1, the mobile analytics topics that are relevant to app artefacts can be organised into themes based on their relationship to the code of the app, the bugs identified in the app, and the app itself. Additionally, there are a set of broader topics relating to trade offs that developers need consider when including analytics in their app artefacts and managing the data pipelines associated with analytics. As with the previous findings chapter, an example of the data analysis performed to derive these themes can be found in Appendix A, specifically Section A.1.

This chapter presents the findings from the case studies together with exploration of grey data and literature, relating to each of these themes. This is followed by a discussion of how mobile analytics affects the artefacts associated with a mobile app project, drawing on the broader literature on how development teams use different types of artefact.
7.1 Code-facing topics

The first broad category of themes relevant to the findings are associated with the artefacts of mobile app projects that are more closely connected to the code of the app. This includes 1) build scripts, 2) integration of the app with mobile analytics SDKs, 3) automated tests for the app, and 4) tests for mobile analytics SDKs. These are followed by introducing one of the ways app developers have used mobile analytics for remote logging.

7.1.1 Build scripts

App developers use and maintain build scripts to build their apps and generally the build script combines with various build tools to customise the Application binary (the code that’s installed on app-centric mobile devices). Commonly used build variants include debug and release build targets. When apps include in-app mobile analytics often the build scripts are modified to specify the necessary SDK dependencies and the application’s source code is modified to initialise the SDK. The documentation for Sentry’s Android SDK provides a clear three step process of Install, Configure, Verify, so these are reproduced here. Listing 1 is a typical example of how a mobile analytics dependency is added to the build file, this is Sentry’s install step in their process.

Listing 1: Example: Install Sentry build.gradle to an Android app’s code-base

dependencies {
    implementation 'io.sentry:sentry-android:6.0.0'
}

source: Android Sentry Documentation
The **configuration** step requires a setting that is uniquely allocated by Sentry for this app when the developers use Sentry’s online service. Listing 2 shows the syntax of the configuration; the developers would need to obtain the unique Data Source Name (DSN) to use for their app and use that value in place of the example value. For authenticated users Sentry provides elegant, dynamic code samples that include the appropriate DSN.

**Verification** is not strictly part of the build scripts as it’s part of the app, nonetheless it’s included here since it is closely connected to the previous steps.

The code snippet in Listing 3 sends details of a caught exception to Sentry’s service each time the code is run. Running this code (by using the app) helps verify the installation and configuration steps have been completed adequately. Note: this code would generally be removed from the app once the verification has been completed successfully, developers would effectively replace it with custom code for any additional reporting not already provided automatically by the SDK.

During the period of this research, the SDKs have continued to evolve, with newer versions offering developers a greater range of information and also more automation of the underlying data collection. For example the Sentry 3.1.0 Android Gradle plugin automatically instruments the OkHttpClient library if it’s part of the app.

The majority of the other SDKs used in the case studies offer equivalent installation, configuration, and verification steps and include developer-oriented documentation of these steps. However, it should be noted that they may use other terms to describe these steps.

**Build scripts encapsulate the build process.** At one extreme build scripts may fully automate multiple steps to the point of releasing a new version of an app in the app store. At the other extreme, many of the steps may be manual and performed unsystematically by the development team. Of course many projects are somewhere between these two poles, so they...
Findings: Apps and their Artefacts

3: StackOverflow has a good example of how a build script obtains these values from the environment in stackoverflow.com/q/46814593/340175. The relevant documentation is no longer available as Fabric Crashlytics was retired by Google in 2020; it was at docs.fabric.io/android/fabric/settings/working-in-teams.html according to stackoverflow.com/a/40667490/340175

[238]: Scott (2017), Android app crash NoClassDefFoundError on Samsung Lollipop devices

partly encapsulate the build process where humans have to perform the rest of the steps in the process.

In the Catrobat case study, a mistake in a manual step of the build process meant that release (0.9.65) of the Pocket Code app did not include the correct information and stopped the in-app crash analytics from being reported. The Crashlytics SDK on Android needed two distinct keys, an API key and a secret key[^3], and presumably at least one of these keys was either missing completely or incorrect for that build.

As an aside, in Grey Data, there is an excellent example not only of configuring the API secret key using the runtime environment (the method used by the Catrobat project) but of an adverse side-effect of changing the build target to a newer release of Android where the app then started to crash frequently on some Samsung device models [238].

7.1.2 Calls to the Mobile Analytics SDK

This research found and identified several distinct uses of mobile analytics by developers. Developers could apply none, any, or even all of these uses in their app for various in-app mobile analytics SDKs.

- **Instantiation only:** a minimal integration that calls the SDK so that it is configured and runs in the background. No other calls are made to the SDK by the app, therefore the only analytics data is whatever is collected by default by the SDK.
- **Reporting of caught exceptions** (also known as Errors and/or error reporting).
- **Breadcrumbs:** some SDKs provide a Breadcrumbs API, for others developers can implement and use custom events to generate an instance of a breadcrumb.
- **Reporting of additional activities:** for example screen and/or network activities. This may be performed automatically by the SDK or by developers writing API calls to do so.
- **Remote logging using a mobile analytics SDK.**

[^3]: Harty et al. (2021), ‘Logging Practices with Mobile Analytics: An Empirical Study on Firebase’

Instantiation Only: In joint research [9] we discovered 50 of 107 active opensource Android apps only initialised the Firebase Analytics SDK. The other 57 made additional API calls to the Firebase Analytics SDK. Pocket Code only initialised Firebase Crashlytics in their Android app (rather than using any additional capabilities offered by the Crashlytics SDK).

Reporting caught exceptions: The C1 project made extensive use of Microsoft App Center to record caught exceptions, in particular; while Moonpig used Firebase Analytics for similar purposes.

Breadcrumbs: Moonpig made extensive use of Firebase Analytics to record breadcrumb information to help determine and understand what led to undesirable events. Project C1 used a proprietary distributed logging API for similar purposes. LocalHalo might have logged similar data using Flurry, their business focused mobile analytics SDK. However,
they did not appear to use Sentry to record breadcrumbs based on the contents of their reports in Sentry.

Grey literature provides code examples and screenshots of creating custom code to add and subsequently use breadcrumbs to an Android app [239].

**Reporting of additional activities:** Both Moonpig and C1 made extensive use of mobile analytics to report additional activities. For Pocket Code some early, exploratory code was written to experiment with recording screens and activities however this work was abandoned with the realisation that Firebase was collecting and providing demographic and other potentially sensitive information from the project team’s perspective.

Logging, network IO, and mobile analytics combined in the industrial case study, C1, where a high crash rate in a new release was caused by a flawed implementation to increase the logging of network IO through the OkHttp library used by the Android app.

Various mobile analytics SDKs automatically record activities and network I/O, a topic for the next chapter.

**Mobile analytics for remote logging:** Some app developers also chose to use mobile analytics for logging, for example 57 active opensource Android projects available on github.com [9].

Mobile Analytics SDKs tend to use a network connection to transmit information from the end-user’s device to central servers. As an aside, it should be practical to write automated tests for mobile analytics SDKs to check the data the SDK emits. Doing so is outside the scope of this research and a possible topic for future research.

---

**Automated tests for logging**

While logging is rarely a source of crashes in mobile apps it’s often used:

(a) by the platform and/or the logging SDK record the actual crash and associated stack trace, and

(b) by the app developers to record information to help the app developers diagnose possible reasons for the crash.

(c) As an aside, Google developers worked with app developers to add logging to help diagnose crashes for some older Android devices github.com/google/filament/issues/2418.

PostHog uses a ShadowLog in their tests of logging github.com/PostHog/posthog-android/.../LoggerTest.java

As mentioned in Section 5.9.1. Experimental apps to increase coverage of mobile analytics services as part of this research I co-wrote various small software projects to provide automated tests of local logging by Android apps [240, 241]; these have been released under permissive opensource licenses for evaluation and use by the research community. These projects may also provide a basis to write automated tests for remote logging using mobile analytics.

---

[239]: Daniel (2019), *Breadcrumbs to enhance your Craslytics’s experience*

[9]: Harty et al. (2021), ‘Logging Practices with Mobile Analytics: An Empirical Study on Firebase’

4: I’m not aware of any exceptions nonetheless other transfer mechanisms are possible such as using memory cards or memory sticks.

5: Crashes can be read from an Android device using developer options and the Android Debug Bridge (ADB) command adb logcat -b crash.

[240]: Reeve et al. (2017), *AndroidCrashDummy GitHub project*

[241]: Reeve et al. (2017), *AndroidLogAssert*
7.1.3 Use of automated tests for the app

Automated tests can dovetail with using mobile analytics to improve the reliability of mobile apps. For example, they can be used to demonstrate the reproducibility of crashes identified through mobile analytics. In the commercial case study this is what we did for various crashes reported via Android Vitals. For one of these an equivalent example has been released as an opensource project which tests the fix. The bug was similar to those discussed in where unexpected data was passed between separate modules of the app. The module that received was written to expect that the data would be complete and correct, however sometimes it was not as some of the data was incomplete or missing. In the case of CI’s app, the mismatched, and incorrect, expectation was detected by mobile analytics and the development team modified the source code to cope adequately without crashing in the event of incomplete or missing data occurring again.

The KotlinNPE project is partnered by another opensource project that replicates the solution the development team sought to run the automated tests for the KotlinNPE project. They wanted to find a way to run tests directly in a JVM without needing the overhead of the Android runtime. When automated tests need the Android runtime the build process and the runtime environment become much more complicated and the elapsed runtime for the tests can take several orders of magnitude more time.

Use of automated tests for bug reproduction/bug fixes was patchy in all of the development teams in the case studies. I.e. they all chose to ‘fix’ at least some crashes directly without the support of automated tests. Their success rates of writing these fixes varied. For example the Kiwix developers were able to fix various NullPointerExceptions directly in the application’s source code whereas they were not able to fix the crashes in the custom downloader code.

Moonpig’s development team chose to write automated tests for bug fixes found via mobile analytics where they believed they would be helpful. For some of the failures they believed they had sufficiently detailed sources of information to apply some fixes directly to the codebase, in part through their extensive use of Firebase Analytics for in-app logging and usage monitoring.

7.1.4 Tests for mobile analytics SDKs

Several types of tests have been found for mobile analytics SDKs, the first type are tests developers implement to check the plumbing works, i.e., that the SDK has been integrated and configured adequately for a basic confidence test. The second type are written to check the SDK at a unit or module level.

Support in the SDK: All of the in-app crash reporting SDKs encountered during the research included confidence tests (verifying the SDK has been configured adequately). The app-centric case studies might have used these verification tests when they enabled their mobile analytics SDKs, however as noted earlier in this chapter these verification tests are
unlikely to be long-term additions to the app’s codebase and no evidence is available on whether they were used or not.

**Crash reporting tests:** The Catrobat project included automated unit tests for crash reporting until the project removed Firebase Analytics from the project several months after the core case study. Kiwix does not use in-app mobile analytics so does not have any tests for them either. None of the interview-led case studies provided details of whether they had crash reporting tests. None of the other app-centric case studies provided any details of whether they had automated tests for the crash reporting SDK.

**Automated tests for the open source SDKs:** Automated tests for the open source SDKs used in the app-centric case studies are available on GitHub: Amplitude Android, Segment.io’s Android SDK, Sentry’s Android SDK and React Native SDK, and Firebase’s Android SDK, amongst others.

While these SDKs have various automated unit tests, end-to-end tests for the SDKs are harder to find. Segment provides an end-to-end test `E2ETest.java`, and clearly describe why they have chosen to test their service in production [243].

7.2 Bug-facing topics

Mobile analytics SDKs are software development projects, and like other software they may have material flaws and some can even stop an app from working. During this research the following items were identified:

- A mobile analytics SDK can cause the app to crash e.g. for Segment IO’s Android SDK Crash during Google Pre-launch report #732.
- Some developers choose to record bugs for failures reported via mobile analytics. (e.g. Kiwix Kiwix Android Issue 2482: for a Crash Report for an IllegalArgumentException in release 3.4.1 of the Kiwix Android app). Note: As part of reviewing this bug I noticed at least one new bug in Android Vitals, where it lists the same crash cluster several times in the results, details are in the issue on GitHub.
- Bug reproducibility, including the use of automated tests (discussed in the previous section).
- The efficacy of addressing bugs (as discussed in the previous chapter on page 153).
- Obviating some bugs by replacing Java code with Kotlin (Upstream improvements). The development teams for Kiwix and C1 both actively pursued this approach, replacing one Java class file at a time with the equivalent Kotlin class file.
- Intermittent bug appearances, covered next.
Intermittent appearances of bugs: In some instances, failures have remained submerged for several releases. Mobile Analytics helps to surface (detect) these failures when they do appear. For instance with the Kiwix Android app, in September 2019 there were 55 crashes reported for a WebViewFactory MissingWebViewPackageException; as shown in Figure 7.2. This crash disappeared for several releases before reappearing. The disappearance might have been for various reasons, one possible reason was simply the particular user who was adversely affected stopped using the app for a while.
7.2 Bug-facing topics

Figure 7.2: Kiwix Android – 55 crashes for one user
7.3 App-facing topics

Three topics in particular focus on the mobile app. The first of these topics - improvements in crash and ANR rates - is the most closely connected to the main research question (see Section 1.3 in Chapter 1, on page 8), in terms of improvements to the product quality.

1. Improvements in crash and ANR rates.
2. Build scripts can affect the behaviour of the app.
3. User Experience of apps with in-app analytics (visibility and control aspects).

7.3.1 Improvements in crash rates

For both Catrobat and Kiwix the hackathon and the post-hackathon bug fixes were highly effective in terms of cumulatively reducing the crash rate over several subsequent releases. The Kiwix hackathon is presented in the first Appendix in A.1. The Catrobat case study replicated the improvements seen in the Kiwix case study and also the efficacy of using a hackathon as an immediate intervention.

For Pocket Code, the improvements in the stability of the app were particularly encouraging as the project had already implemented many of ‘good’ development practices. Figure 7.3 shows the cumulative improvement in the crash rate as measured by Fabric Crashlytics. Two releases to Pocket Code were made with fixes for issues identified during the hackathon.

Note: Before the hackathon they had already addressed one of the worst offenders in terms of the crash rate and the effects of that improvement were rolling out during the weeks after the hackathon. 16 was the #1 issue with over 2000 crashes reported in the 30 days preceding the hackathon (and over 3200 in the lifetime of the app). However, it had already been addressed in Jira issue 379 and incorporated in to the most recent release 18 as part of a major effort to improve the code quality around the embedded WebView component.

Early indications are that the fixes related to the WebView have made a material improvement in the crash-rate for Pocket Code 2.45% vs. 3.93% on Monday 18th November 2019; details are in Table 7.1. Therefore, in terms of assessing any improvement that results from the work of the hackathon the baseline is 2.45%.

Some notes on Tables 7.1 and 7.2 (and on interpreting crash and ANR rates in Android Vitals generally):

1. The data was reported for thirty-day periods, one of three durations available in the user interface. All figures were calculated and provided by Google’s Android Vitals analytics.

<table>
<thead>
<tr>
<th>App version</th>
<th>Impacted sessions</th>
<th>Crash-free sessions</th>
<th>Number of sessions</th>
</tr>
</thead>
<tbody>
<tr>
<td>69 (0.9.65) Production</td>
<td>2.45%</td>
<td>97.55%</td>
<td>800</td>
</tr>
<tr>
<td>66 (0.9.64) Previous release</td>
<td>3.93%</td>
<td>96.07%</td>
<td>3K</td>
</tr>
</tbody>
</table>

Table 7.1: Android Vitals: Improvement in crash-rate for Pocket Code post WebView improvements, as recorded on Monday 18th November 2019
7.3 App-facing topics

Figure 7.3: Fabric Crashlytics report on improvements in Pocket Code’s crash rate

Table 7.2: Android Vitals: Improvement in crash-rate post Hackathon

<table>
<thead>
<tr>
<th>App version</th>
<th>Impacted sessions</th>
<th>Crash-free sessions</th>
<th>Number of sessions</th>
<th>Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>72 (0.9.68)</td>
<td>1.26%</td>
<td>98.74%</td>
<td>9K</td>
<td>11-Jan-2020 to 08-Feb-2020</td>
</tr>
<tr>
<td>71 (0.9.67)</td>
<td>1.25%</td>
<td>98.17%</td>
<td>14K</td>
<td>11-Jan-2020 to 08-Feb-2020</td>
</tr>
<tr>
<td>70 (0.9.66)</td>
<td>1.65%</td>
<td>98.35%</td>
<td>5K</td>
<td>04-Jan-2020 to 01-Feb-2020</td>
</tr>
<tr>
<td>69 (0.9.65)</td>
<td>2.05%</td>
<td>97.95%</td>
<td>2K</td>
<td>22-Oct-2019 to 19-Nov-2019</td>
</tr>
<tr>
<td>66 (0.9.64)</td>
<td>3.61%</td>
<td>96.39%</td>
<td>16K</td>
<td>22-Oct-2019 to 19-Nov-2019</td>
</tr>
</tbody>
</table>

2. The reports exclude small aggregate counts that fail to meet Google’s reporting thresholds.
3. The percentages fluctuate depending on where releases are in their maturity (their production lifecycle). Therefore these values are spot figures calculated that day and indicative rather than necessarily being suitable for calculating ratios for different periods.
4. These reports appear to be regenerated once a day, roughly 24 hours apart.
5. It is unclear whether the underlying data is filtered to limit it to apps only installed by end users of production releases from Google Play.

Table 7.3, on page 171 provides a comparison of the crash rates for the experiment and the control apps over four stages of this case study. There were three phases that interleave with these four stages.

The stages are:

1. The baseline,
2. Simplifying the most buggy code (which was the downloader),
3. Applying the concepts in the research to the experiment,
4. The new normal when mobile analytics is actively used.

With the CI industrial case study the crash rate was reduced fourfold and the ANR rate was reduced twofold. Furthermore, when a developer inadvertently introduced the exception that led to a short term crash rate of over 10%, caused by a flaw logging OkHttp network responses, the project team were able to quickly detect and address the cause. One the updated release was rolled out the crash rate returned to approximately the previous mean value.

19: The actual values are confidential.
Moonpig provided various details of two crashes that emerged in production, including several graphs. Figure 7.4 illustrates the effects of the crashes where they adversely affected the crash-free session scores and shows the restoration of higher reliability as each crash was addressed.

7.3.2 Improvements in ANR rates

The Catrobat project was able to improve the ANR rate in several releases after the hackathon. Table 7.4 shows the improvement in the ANR rate three days into the rollout of the new release (0.9.66). (This release also improved the crash rate as Table 7.2 shows.)

Further improvements included Catrobat’s Pocket Code app, where ANR at org.catrobat.catroid.stage.StageListener.takeScreenshot (StageListener.java:664) was fixed three months after the hackathon and merged into v0.9.70 of Pocket Code in github.com/Catrobat/Catroid/pull/3402.

Several of the case studies experienced bursts in the ANR rates for at least one of the Android apps. For the CI project, the ANR rate in one release escalated to approximately 10% in a rogue release before being reduced steadily to less than 0.4%, partly by replacing an outdated version of an in-house library developed by another part of the organisation.

In grey material, a company valued at approximately 5 billion USD, Bumble, achieved a six-fold reduction of ANRs by actively choosing to do so. They implemented hybrid approaches to scrape the content of reports from Android Vitals, this was - in essence - similar to our work on Vitals Scraper however they used Selenium rather than TypeScript, and Vitals Scraper obtained a wider range of data and the individual failure logs. They also wrote an in-house library that uses Android’s getHistoricalProcessExitReasons to obtain details of the ANRs.

They used a multi-faceted approach to reduce the ANRs, described in [245]. In short they used data, metrics and analytics to inform their development work and they were highly effective in achieving the desired reductions over a series of releases of their Android apps.

7.3.3 Build scripts can affect the behaviour of the app

A previous section Section 7.1.1. Build scripts explained how build scripts encapsulate the build process and how it’s possible to adversely affect the behaviour of in-app mobile analytics. And in Section 6.1.3. Choosing mobile analytics of the previous chapter, how developers can choose to vary the behaviours of an app, e.g. to increase the logging in debug builds. (In Android they can use ‘build variants’ and ‘product flavors’ to do so [246].)

Often developers wish to optimise the application binary to remove non-essential code and to obfuscate the object code within the binary to protect the intellectual property contained within. Both the optimisation and any obfuscation tend to occur for release (production) builds of an app, these may affect the behaviour of the app in ways end-users are hard-pressed to discern. When using in-app mobile analytics it’s important to preserve this functionality in release builds. Some projects
7.3.4 User Experience of apps with in-app analytics

The visibility of any end-user ‘agreement’ varied across the apps in the app-centric case studies that used in-app mobile analytics. Most of the apps did not present users with this information, with the exception of Pocket Code. It required end-users to read and agree to a lengthy legal agreement the first time they used the app.

Informed consent: is an awkward conundrum for both end users and app developers as neither is particularly interested in the minutiae when compared to the functionality of an app. At the time of the Catrobat case-study their Android app opened with a mandatory screen where users were faced with the following text in Listing 7.1:

Welcome back, 727 words 4,590 characters later. This example of an app privacy policy here provides an idea of how adding a mandatory, relatively comprehensive privacy policy interrupts the flow. Few users would voluntarily read these terms, and, via Firebase Analytics, we discovered that the Pocket Code app only retained 4% of new users.

Welcome!  

1. Welcome!  
2. If you want to start coding, please read and accept our Privacy Policy to use the app.  
3. To offer you all the benefits of our services and the associated ecosystem, it is necessary to collect certain data from you. 
4. To maintain and improve our services (apps and websites) and to do scientific research on ICT and STEM/STEAM education, we use Google Analytics, Crashlytics, and Firebase, AIS by Google LLC (USA), as well as Dynatrace by Dynatrace LLC (USA) to get metadata about the usage of our services, e.g., crash information, timestamps, visited pages/screens, usage of our apps and services, used operating system, and information about the network provider. This analysis and the collected data is not linked to your profile and does not contain any personal information (last bits of the IP address get anonymized). However, the collected data will get transferred to the service provider (Google LLC, USA and Dynatrace LLC, USA). 
5. Data processing is based on agreements with Google LLC and Dynatrace LLC. 
6. This relationship to the service provider conforms to the European Union’s General Data Protection Regulation (GDPR) and EU-USA ‘Privacy Shield’ agreement. 
7. To create and use an account in our systems, and to inform you about necessary changes about your account per e-mail, e.g., updates or violations of our Terms of Use and Service, our community rules, or our policies, we will store your information and, if provided by you, your e-mail address. If you have connected your account via your Google or Facebook account, we additionally store an identification key connected to your account on these services. We will not see this data for any other (e.g., marketing) purposes, unless you explicitly allowed us to do so. 
8. In a voluntary basis, you can give us your country of residence through your account page on the sharing site. This will get displayed on your public profile and only be used for statistical and research purposes by us. It can be removed at any time in your profile settings. 
9. You can withdraw the usage of the personal data belonging to your account at any time by deleting your account through your profile page on https://omega.catrobat.at. After deleting your account you will still be able to use the provided services, but not to collaborate, e.g., through uploading programs, commenting, or liking. Also, all provided content linked to your account, e.g., uploaded programs, comments, etc., will be deleted. 
10. To avoid misuse of our services, e.g., through user contributed uploads or comments for illegal purposes, all user generated public data (uploads, comments) will be stored internally in a database, together with a timestamp and the used internet address. This data will not be used by or forwarded to any institutions, unless sufficiently illegal actions are reasonably suspected and officially entitled legal institutions request it based on applicable law. Each case will be thoroughly checked on an individual basis by us first. 
11. You can voluntarily subscribe to the Catrobat Newsletter, on https://catrobat.at/newsletter provided by us through the MailChimp service (The Rocket Science Group LLC &/or MailChimp, USA). You will then receive updates on the Catrobat project and its services to your provided e-mail address. You can unsubscribe the newsletter at any time by using the ‘unsubscribe’ link in the footer of every newsletter. This newsletter is provided by MailChimp with whom we do have a data-processing agreement and who committed to the EU General Data Protection Regulation and EU-USA ‘Privacy Shield’. For further details on MailChimp please also look up MailChimp’s Privacy Policy and Terms of Use. 
12. All Services are provided by the free open source Catrobat project under the umbrella of the International Catrobat Association – Verein zur Förderung Freier Software, a non-profit NGO incorporated in Graz, Austria (European Union). Our policies pay respect to EU and Austrian data protection law (DPA). To get in touch with us, please send an e-mail to contact@catrobat.org or mail us to Catrobat, c/o Institute of Software Technology, Graz University of Technology, InfraLinguage 1B, and A-8022 Graz, Austria (European Union). 
13. In case you are below the age of 14, this policy must be accepted by a legal guardian (usually a parent). 

Listing 7.1: Privacy Policy for Pocket Code
by day 2 which indicates many may simply abandon the app when presented with these terms at startup.

In contrast Moodspace took a light-hearted approach in their app, as follows:

<table>
<thead>
<tr>
<th>Moodspace privacy policy</th>
</tr>
</thead>
</table>
| “To make the app work well at all we collect the following anonymous data:

- **Crash reports:** If you’ve never seen the app crashing, it’s because as soon as one happens, we get a crash report. A little red light flashes in our office, a loud siren blares, and we release a fix right away. It’s quite annoying actually.

- **Analytics:** We assume you’re going to use the app a certain way. We’re almost always wrong, and you often surprise us. Analytics lets us see how people like you actually use the app, so we can make improvements to the right places. Analytics can use the Google Advertising ID to identify you. This doesn’t tell us anything about you (it’s just some numbers and letters), but if you really want to trick us you can reset your Google Advertising ID at any time. Go to your device Settings > Google > Ads.”

[249]: Moodspace (2021), Privacy Policy  | Moodspace

None of the apps provided end-users with any mechanism to disable in-app mobile analytics.
### 30-day crash rates reported in Android Vitals

<table>
<thead>
<tr>
<th>Stage</th>
<th>Release</th>
<th>Experiment</th>
<th>Control</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Kiwix</td>
<td>WikiMed English</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>5.07%</td>
<td>1.13%</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3.12%</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>2 2</td>
<td>1.59%</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0.53%</td>
<td>1.09% 5</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0.72% 6</td>
<td>0.60% 7</td>
</tr>
<tr>
<td>4 8</td>
<td>4</td>
<td>0.55%</td>
<td>0.41%</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>0.40% 9</td>
<td>0.26% 10</td>
</tr>
</tbody>
</table>

0 Except when otherwise noted.
1 Kiwix Release 2.5 with the previous custom download facility replaced by a Google Android downloader.
2 The code is under 25 lines including 10 lines of comments [https://github.com/kiwix/kiwix-android/pull/1388](https://github.com/kiwix/kiwix-android/pull/1388).
3 Aggregate crash rate over 7 days for versions 2.4, 2.5.1, 2.5.2, 2.5.3 (to Aug 26th 2019).
4 Previous 30 days crash rate, before release 3.1.2 pushed the crash rate up (same graph as TODO).
5 *Unchanged release from the first control.*
6 Includes 3.1.2 which had an average (mean) crash rate of roughly 1.7% (roughly 31st Dec 2019).
7 A mixed set of crash rates, averaged by Android Vitals. For the first updated release of Wikimed (the 2019-12 release).
8 As usage increased of the more reliable releases the averages declined.
9 The crash rates for releases 3.2.1 are 0.23% and 3.3.1 are 0.31%.
10 Release 2020-03 actually has a crash rate of around 0.05% the numbers are higher as there are still significant volumes of usage on the previous 2 releases.

#### Table 7.3: Improvements in Kiwix Crash Rates for the main Kiwix app and WikiMed in English

<table>
<thead>
<tr>
<th>App version</th>
<th>Impacted sessions</th>
<th>ANR-free sessions</th>
<th>No. sessions</th>
</tr>
</thead>
<tbody>
<tr>
<td>70 (0.9.66) Production</td>
<td>0.30%</td>
<td>99.70%</td>
<td>700</td>
</tr>
<tr>
<td>69 (0.9.65)</td>
<td>0.46%</td>
<td>99.54%</td>
<td>3K</td>
</tr>
</tbody>
</table>

#### Table 7.4: Pocket Code: ANR rate for Last 7 days, as recorded on 1st December 2019
7.4 Macro topics

Macro-topics touch on multiple aspects of developing mobile apps. This research identified the following groupings of macro-topics:

▶ Three of these have a common trait of **tradeoffs** developers face, in ethics, in the use of third-party code, and in functionality.
▶ Two have a common trait of **scaling**: scaling use of mobile analytics by developers and within their engineering microcosm.
▶ Two cover the trustworthiness and fidelity of the mobile analytics as reflected in the artefacts.

7.4.1 Tradeoffs

At least three types of tradeoffs were evident in the artefacts:

1. **Ethical tradeoff**: the effects of ethical decisions by the project team (covered in Section 6.1.4. Ethics and Personally Identifiable Information (PII)),
2. **Functionality tradeoff**: downgrading functionality to obtain reliability,
3. **Third-party tradeoff**: use of third-party vs locally developed functionality.

**Ethical tradeoffs**: (Tradeoff 1). The clearest example from the case studies, as described in the previous chapter in Section 6.1.4. Ethics and Personally Identifiable Information (PII), is where the Catrobat project chose to remove in-app analytics from Pocket Code to prevent data being collected by Firebase Analytics via this app as other apps on the end user’s devices almost certainly are using Firebase Analytics and reporting much or all of the demographic and geographic data to Google’s systems.

**Reliability trumps enhanced functionality**: At the start of the Kiwix case study, the core Kiwix Android app included a custom file downloader that downloaded material such as Wikipedia in German for offline use. This downloader kept track of partial downloads and also allowed end users to control when to allow the downloads and when to pause or stop them. This functionality was designed to help users often in areas with intermittent, controlled, and sometimes expensive connectivity to increase their chances of being able to download the content within their budget. To provide some concrete examples in some parts of the world users would download content that took several entire days even if the connection didn’t fail, and the cost of the connection was metered and capped to 1GB of data.

However, the code for the downloader had various bugs and flaws including several that caused the app to crash. These crashes meant the app had an excessively high crash rate in the Google Play App Store. The then lead developer for the Android codebase had not been able to tame the high crash rate. A consultant was hired who had worked on multiple other Android projects and codebases. This consultant proposed scrapping the custom downloader and replacing it with standard Android functionality provided by Google as part of the Android SDK. This
7.4 Macro topics

The proposal was accepted and implemented and it had the desired effect of reducing the crash rate of the core Kiwix Android app significantly, as illustrated in Figure 7.5. The headline reduction in the report is from 4.59% to 3.12% however this includes the interim period when the reduction was taking effect as more users received/installed the newer release, so the improvement was better than this report suggests - roughly a reduction from over 4% to 2%. (The gap in the graph is one of the flaws discussed in the tools chapter, in Section 8.4.1. Flaws in the mobile analytics tools and/or services.)

However the standard Android downloader didn’t provide equivalent facilities to continue partial downloads, or to pause and resume downloads, nor did it provide progress tracking information. Furthermore as the project team had chosen not to incorporate in-app mobile analytics the project team cannot easily determine the effects of these changes on the end users. Bugs continue to surface occasionally on the project’s GitHub site and users sometimes complain in reviews submitted to Google Play. Here are four verbatim examples extracted from the developer’s view of Google Play; (the developer’s view has greater visibility into ratings and reviews than the public (user-oriented) view):

▶ “Wikipedia nopic without images works again in version 11.18 with search function! Top Unfortunately, now in March 21 again problems with wikipedia text with 14.2 gb. After 3/4 of the download the process breaks off every time! Too bad. Wikis with smaller amounts of data can be downloaded without any problems. Handling of the app very well.”, Mar 22, 2021
▶ “Download started and it could never finish”, Mar 25, 2021
▶ “Seems that library download is borken atm?”, Oct 7, 2021
▶ “Cant continue interrupted download”, Oct 15, 2021

And another review asks for the download manager that, ironically, used to exist (presumably without realising the app used to have one): “edit: still there is no download manager. It is not possible to download the large files. Despite 100 megabits per second, I did not manage to load the large Wikipedia with images. The app itself is very good. Only the download of the huge files always breaks off. I load via PC. A download manager would not be bad.”, Mar 12, 2021

Figure 7.5: Kiwix crash rate drops once custom networking code replaced

---

24: e.g. github.com/kiwix/kiwix-android/issues/2845.
Third-party code: The third tradeoff is between third-party and locally developed code which generalises the second tradeoff found in the research (of trading functionality for reliability). As a concrete example, the WebView component is ubiquitous on Android and found in many Android apps including several of those in the case study.

As projects, including the Kiwix team, discovered crashes can be reported in Android Vitals for the WebView component. These adversely affect the app’s reliability as measured by Android Vitals. While the app developers can fix at least some of the causes of third-party components, including the WebView component, others may be beyond their direct control.

In March 2021, major apps including apps from the Amazon, BBC, Facebook, Google, Microsoft, etc. failed in use by a release of the WebView component [14, 15, 250]. and Samsung’s deadpan observation:

> “If you are having an issue with the apps forcibly closing on your device, please follow the steps to resolve the issue. This problem has been resolved with the latest app updates of Android System Webview and Chrome, 89.0.4389.105 version.”

> “To ensure that your apps do not crash, please update both Google Chrome and Android System Webview.”

All their respective app developers relied on a third-party software component developed and maintained by Google as part of Android. Furthermore, app developers who have incorporated the WebView component into their app sometimes introduced bugs in their use of the WebView. Across a population of 146 opensource Android apps 124 WebView related bugs were found and the root causes analysed [251, pp. 704 - 706]. Of these bugs, 15 introduced crashes [251, p. 706].

As Google Android notes online https://developer.android.com/guide/webapps there are various alternatives to using a WebView. They don’t mention either a) developers developing their own user interface, or b) other third-party alternatives to their WebView (e.g. Mozilla’s GeckoView).

There is no silver bullet for developers in terms of choosing whether to write their own code or use third-party code. A functionality tradeoff (tradeoff 2) enabled the Kiwix team to improve the reliability at the cost of losing functionality where they replaced their own code with third-party code.

The third-party tradeoff (tradeoff 3) allows developers to short-circuit the work of developing their own functionality at the risk of being unable to address some of the failures and problems related to using those third-party components. Examples from the app-centric case studies include the use of RoboSpice by Moonpig, and the Expo framework by LocalHalo.

### 7.5 Improvements to the artefacts

Developers and their wider development teams should expect to encounter flaws from time to time on an ongoing basis and also accept...
the need to address these through improving the apps and their software artefacts accordingly. Similarly they can use mobile analytics in greater depth and breadth in order to increase the amount (and timeliness) of information they receive. Finally for this section, preserving reports and results can help the team work more efficiently and effectively.

### 7.5.1 Improving the source code

This might seem trite, or obvious, nonetheless the reported failures indicate material flaws in someone’s source code, whether that’s source code of an app, a software library, automated tests, build scripts, platform SDKs etc.

Several approaches were used in the app-centric case studies, such as developing automated tests that reproduced a crash, discussed in Section 7.1.3, replacing software libraries by Kiwix and Moonpig covered in the previous section, and replacing Java code with Kotlin covered earlier in Section 7.2.

### 7.5.2 In-app crash analytics

In-app crash analytics are seen to provide valuable information by the case-studies who have incorporated them. They complement, and are complemented by, the platform-sourced mobile analytics. Therefore, for developers who have decided in-app mobile analytics abides by their ethical requirements they can select in-app mobile analytics to obtain an in-app perspective of Technology-facing [252, discussed in Chapter 6] information.

Where the developers invest the time to analyse what the SDK does and does not collect they are able to make a more informed decision, rather than later facing the unenviable choice of ceasing the use of their chosen mobile analytics service. The experience of the Catrobat team with the transition to Firebase Crashlytics exemplifies this happens.

Several development teams, including C1 and Bumble [244], chose to implement in-house mobile analytics which gives them greater control on who has access to the collected data; and product offerings from Sentry and others offers a similar level of control though a predetermined core set of data which keen developers could choose to augment. As an example [253] is the data collection module for Countly’s Android SDK, and released under the MIT license so modifications can be private or public.

### 7.5.3 Logging errors

When the app includes a Technology-facing in-app mobile analytics SDK the SDK generally provides an API developers can use to log errors. These errors generally include caught-exceptions which the app has handled; nonetheless developers can generally choose to report pertinent information provided it matches the calling interface of the SDK. These errors often indicate aspects including those beyond the immediate app where the quality of service to the end user may be
improved (amongst other things). For example, several of the apps in the case study interact with servers on the internet where network requests may receive responses indicating problems, whether ephemeral or more systemic. By recording the network responses, including those in the error categories the developers can choose to refine their application logic to minimise the effects of these errors. If they are also responsible for the respective internet servers (for instance, Kiwix, Catrobat, CI, and Moonpig) they may be able to improve those too.

7.5.4 Using breadcrumbs

Similarly, many of the Technology-facing in-app mobile analytics SDKs include API calls that can be used to record Breadcrumbs, these breadcrumbs are surfaced iff a crash occurs subsequently. One of the many challenges faced by app developers is bug reproduction, Section 6.4.2. Bug reproduction, and breadcrumbs may facilitate bug reproduction for those crashes.

7.5.5 Distinguish analytics for each release zone

Distinguishing between analytics from various release zones (e.g. for internal test, pre-release, and production) enables the development teams to track the reliability in each zone independently and without polluting the production analytics. The CI project, used this approach successfully. To apply this approach using in-app mobile analytics the development team need to first create distinct configurations, this is often performed in the user interface of the mobile analytics service as it generally needs to be configured to receive and accept the distinct configuration keys, etc. The development team then need to configure their build system to select and use the correct configuration key(s), for the respective release zone. For Android apps, this is often performed as part of the Gradle build settings.

7.5.6 Preserving reports and results

As illustrated by the app-centric case studies, teams varied in how often they recorded the reported issues in their bug tracking systems. They also varied in how much information they recorded. The reports are often impermanent and details of the failure may fade from the developer’s memory too. Therefore, preserving reports and results (and streamlining the tools and processes to do so) can help improve the bug tracking aspects of the project and potentially also help find longer-term patterns and more systemic issues.
7.6 Chapter summary

This chapter presented the findings concerning the apps and their related artefacts, with examples from the various app-centric case studies. These findings were augmented with analysis of 107 open source Android apps to understand how those developers used Firebase Analytics for remote logging purposes.

► **Code-facing:** Build scripts encapsulate the integration of any in-app analytics SDKs. The integration process tends to be well-documented by providers of mobile analytics, and many include instructions on how to test the integration. Automatic crash reporting is an intrinsic feature of integrating SDKs with this capability; error reporting, breadcrumbs, and custom data collection require devs to write code to call additional APIs. Many of the SDKs also collect additional data such as network I/O automatically, so developers can receive a great deal of potentially-useful information for small amounts of effort. Automated tests for the in-app mobile analytics SDK were devised by Catrobat; some mobile analytics providers provide various automated tests for their SDK.

► **Bug-facing:** Mobile analytics SDKs have bugs, and some can cause the app to crash – inadvertently being worse than not using them at all. Some classes of bugs in the app can be obviated by changing programming language, for Android apps from Java to Kotlin. Some fatal bugs (e.g., crashes) appear intermittently in the reports; it’s not safe to assume they are fixed just because they don’t appear in the report currently.

► **App-facing:** Mobile analytics helps developers find bugs that affect end users, even bugs that have eluded other ‘good practices’ in software development, such as extensive coverage by automated tests. The fixes are often relatively straight-forward and effective. Developers must be aware that build scripts affect what is built and may affect the behaviour of any in-app mobile analytics. Informed consent of end users is an unsolved challenge in terms of mobile analytics.

► **Tradeoffs:** There are ethical, functional, and third-party tradeoffs. Developers are responsible for ethical implications of the data that is collected. Reducing functionality may improve reliability; however, some users may miss functionality that has been removed to improve reliability. Third-party code can cause some of the failures in an app.

► **Improvements:** Developers improve their source code by applying the results of mobile analytics. Adding in-app technology-facing mobile analytics provides information that developers prefer to the default platform-level analytics, even if it lacks some of the failures, as the contents are more actionable. Some app developers create their own mobile analytics that give them greater control. Preserving reports and results is important for the longevity of the information and helps the wider development team.

Developers were consistently able to effect material improvements in the reliability of their apps, as measured by mobile analytics. The amount and complexity of changes to the source code varied from a few lines
of straight-forward code, to major surgery to the app’s codebase, for instance when functionality was replaced.

The various improvements to the apps and their artefacts included how analytics could improve the detail of information recorded in error messages and logged issues, as well as the ability to track the behaviour of different releases by configuring build processes appropriately and preserving analytics reports. These were covered in Section 7.5. Improvements to the artefacts.

The next chapter presents features of the mobile analytics tools.
This chapter covers the last two of the six perspectives, i.e. using and improving mobile analytics tools (uTools and iTools). The primary evidence comes from both the app-centric and the tool-centric case studies and introduced in Chapter 5, augmented with material from grey data and grey literature.

The evidence has been analysed and prioritised to keep the chapter relatively succinct and on topic. In total 38 discrete themes (L1 themes) emerged in the analysis of the evidence, of these the 18 with strongest support in terms of the evidence are included here, the rest would benefit from further work. As before, an example of the analysis performed to derive these themes is presented in Appendix A, specifically in Section A.2. Crashes in cross-platform containers. The associated findings are described in Section 2. Runtime encapsulation of failures.

The L1 themes included in this chapter have been aggregated into four higher-level L2 themes: Section 8.1. Design, Section 8.2. Fitness-for-purpose, Section 8.3. Utility, Section 8.4. Dependability. Figure 8.1 illustrates the top L1 themes and their primary higher-level (L2) theme.
8.1 Design

Design emerged as by far the most pertinent topic for the mobile analytics tools. There are two key, connected facets:

▶ the design of the on-device SDK, including addressing various engineering challenges and deciding on the meta-data to collect; and
▶ UX design to engage the developers to actually use the results of the mobile analytics [effectively].

The design of the on-device SDK is important because any in-app SDK needs to integrate easily in to the mobile app; and platform-level analytics need to be seamless and collect sufficient pertinent information to be useful for the app developers. They also need to be robust and timely in terms of collection, transmission, and processing of the underlying data in order for developers to have timely access to the results.

Mobile Analytics tools need to be used to be effective, and the user experience of the developers who use these tools where “developers’ needs are characterized by efficiency, informativeness, intuitiveness, and flexibility of the tool.” [254, p. 104]. Where using a tool is rewarding for the developers they are likely to use the tool more [255, p. 260].

These tools are a subset of software trying to get a developer’s attention and they need to fit within a larger context. The tools need to surface (make

---

visible) functionality and capabilities that align with the motivation(s) of the developers [256, p. 2].

Fabric Crashlytics is an archetypal example of how a mobile analytics tool can be designed to serve developers well. The product team developed it from the ground up, starting with excellent crash reporting, to provide developers with timely, actionable, attractive, and useful reports. This led to it becoming one of the top three mobile analytics tools for both iOS and Android within 10 months of being launched [257]. First Twitter acquired it and then Google did; they subsequently integrated it into Firebase Analytics which is the most popular mobile analytics service for Android apps currently.

It exemplified good design in terms of the SDK as it collected pertinent data developers found useful without requiring significant effort by the developers. The development team who created the SDK and the product had used their frustrations from using other analytics software as a catalyst to create Crashlytics.

Similarly the user-interface of Crashlytics was slick from the outset and quickly adopted by app developers through presenting the analysis of the data the SDK had collected. It was designed from the outset to be actionable save ‘developers from information overload or “analysis paralysis’” [258].

Developers found it useful, it was free of charge, and the product continued to evolve and improve rapidly, for example by adding a general purpose mobile analytics service, called Answers, to Crashlytics that, in their words: “Before Answers, developers had to wade through mountains of data about their apps to find what they were looking for. We wanted to fix this, so we went to the drawing board and set out to build a mobile analytics solution you didn’t need to analyze.” [257]

Platform-level analytics provides an outsider’s perspective on the behaviour of mobile app. This is in contrast to in-app analytics, which provides an insider’s perspective. In this research Google’s Android Vitals has been the focus of investigation for the platform-level analytics as it has the largest reach of any platform-level analytics service across the widest range of devices. The Google Android platform provides users the ability to allow or deny analytics to be sent from their device. Apple’s iOS (and MacOS) ask the user explicitly [259], Android does not – users need to find the setting and opt-out [260].

Mobile Analytics tools vie for attention against a plethora of other developer-oriented tools, project demands, etc. Developers need to be enticed into using the tools and then retained on an ongoing basis to meet the objectives of the providers of the mobile analytics services.

8.1.1 SDK design

Any mobile analytics SDK needs to be designed to collect relevant data and forward that data so it can be processed, analysed, and reported on. The design of the client-side SDK affects many aspects of the data collection which then feeds subsequent stages in the processing of the data to provide the mobile analytics. Some of the general features of SDK design include the support for different programming languages,
initialisation, and automation of data collection. Additionally, we need to consider how the SDK encapsulates runtime failures, manages metadata about the end users’ devices, alongside other engineering challenges. In this section we draw on evidence from both app-centric and tool-centric case studies to discuss current use and potential improvements to mobile analytics tools, across these different aspects of analytics SDK design.

**Programming language support:** Mobile apps can be written in several programming languages, including Java, C++, Kotlin, React Native, and others. While many mobile apps are written in a single programming language some use several programming languages, for instance Kiwix Android combines Java, Kotlin, and C++.

Mobile analytics SDKs, in turn, support one or more of the programming languages. If they do not support the programming languages then they may not be able to obtain or provide analytics for elements written in the unsupported programming languages. For C and C++ code in particular, the app developers generally need to explicitly configure the code and the build process to incorporate the relevant mobile analytics SDK if it’s available.

**SDK initialisation:** The SDK needs to be initialised as early as practical each time the app is started (or restarted) if it’s to capture pertinent information (including crashes that occur when the app starts or restarts): “for these products is that it would have to be wired in super early in the App’s lifecycle, to (say) allow Crashlytics to capture crashes that happen early on” [261, issuecomment-635498836]. For mobile analytics SDKs this has led to the developers of the SDK finding and implementing mechanisms to initialise their SDK in innovative (and unusual) ways, for instance Firebase uses a Content Provider [262]. Note: this does not always work, as reported in [263]. When the SDK initialises it obtains various meta data about the app and the device.

**Data automatically collected by SDKs:** Mobile analytics SDKs have collected data automatically for years, the developers do not need to write additional code to collect this data. The data includes meta-data about the device and version of the platform. Depending on the SDK it may also collect demographic data, sensor data such as the geo-location, other data such as other apps that are installed, and various events that occur including network requests and responses. The data collected is covered shortly on Page 185.

Any of these data elements may help developers to improve their software; however, use of this data may be considered a privacy risk, particularly for end-users and may lead to ethical conundra for the development team and their organisation, as discussed in Chapter 6 in Section 6.1.4.

**Breadcrumbs:** In 2015 HP’s launched AppPulse Mobile SDK automatically instrumented a mobile app to add crash reporting and the recording of screen transitions together with various meta-data. Both their iOS and Android SDKs provided similar capabilities in terms of an API developers could optionally call to record breadcrumbs in: Android [264, 265] and iOS [266].
Runtime activities for the SDK: When the SDK is running, which they do in the background without being visible to the user of the app, they are responsible for the safekeeping and transmission of the collected data. Some collect data automatically, or autonomously. For example, Sentry’s in-app SDK collects ‘automatic instrumentation’ [267], and Android Vitals collects usage data, app crashes, and ANRs automatically.

At least some of the SDKs store analytics data locally on the device on an interim basis, the stored data would be removed once it had been successfully transmitted. Various SDKs limit the number of items they store, for example the default event store for Microsoft App Center on Android is 10MB [268]. The SDKs also vary in how and when they transmit the data and on their behaviour if there isn’t a suitable network connection to transmit the data. Microsoft App Center’s documentation is clear and a good practical example worth study [Ibid].

In-app analytics support for detecting ANRs: At the start of the research in-app analytics SDKs were not able to measure ANRs which meant Android Vitals was the primary source of ANR analytics for Android app developers. Subsequently, an open source utility called ANR watchdog was released that uses a watchdog timer to detect ANRs [123]. Investigating it in depth was beyond the scope of the immediate research, nonetheless Sentry used that code as a basis for their ANR reporting (sentry-android-core...ANRWatchDog.java).

Note: Google has subsequently added a mechanism to enable apps to obtain information about previous ANRs when the app next started. The method is `getHistoricalProcessExitReasons()` 1, added in Android 11, API level 30. At the time of writing, Firebase Analytics uses this mechanism to obtain the ANR and other app exit data 2.

Runtime encapsulation of failures

The choice of runtime can affect what various mobile analytics can detect in terms of errors and failures in the app. Various details of this example are covered in Appendix A in Section A.2. Crashes in cross-platform containers.

Limitations in visibility by an SDK: In short, the viewpoint of the SDK affects and can limit what it can observe/record. Also some mobile apps incorporate their own runtime which may hide some failures from being observed by the platform.

Android Vitals: Android Vitals does not collect crashes that are contained within an application's runtime. React Native is a popular cross-platform app development framework. It includes its own application runtime environment and this runtime automatically restarts the app if it crashes. These crashes are not visible to Android Vitals as evidenced by two of the apps within the app centric case studies – LocalHalo and GTAIF’s Taskinator app – where Android Vitals showed no crashes for either of these apps, with one exception.

The LocalHalo app-centric case study provides an illustration where app crashes were not observed by Android Vitals until a failure in the React

[267]: Hofmann (2023), Mobile Vitals - Four Metrics Every Mobile Developer Should Care About

[268]: The contributors (2023), App Center Analytics (Android)

Native runtime occurred. Note: the data collection, analysis, findings, and conclusions for this example are presented in Appendix A in Section A.2. Figure 8.2 reflects the empty reports typically provided by Google Play Console for the two apps written in React Native covered in this research: LocalHalo and GTAF.

Figure 8.2: No Android Vitals reports on 16th March 2020

Figure 8.3 was recorded ten days later in 26th March 2020 and shows the alerts for both high crash and ANR rates in the App Health Overview page and the graph for the rampant crash rate in the corresponding App Health Details page. These indicate the failures were related to the native runtime rather than within the React Native code. These were not reported by any Sentry Alerts and they do not appear in the weekly summary reports, except potentially by the absence of data shown in Figure 8.4. While the reason for this was not explained in the interviews or in the analytics data, it is likely that this caused by severe crashes that prevented Sentry’s SDK from reporting any data.

A release in March 2020 had a high crash rate for the production release of their Android app. The top crash cluster was for:

```
java.lang.RuntimeException
```

This was traced to a problem in the Expo library the development team used in the app [269][3]. In that issue, several developers for different Android apps provide data from Google Play Console confirming they also receive similar crash clusters. The cause was not definitively traced or addressed by the LocalHalo development team; however, for the LocalHalo app the crashes stopped being reported once a new release (1.3.0) of the Android app was launched around 6th April 2020.

Some failures did emerge when the runtime encapsulation fails: That exception was when Android Vitals did report crashes in March and April 2020. Figure 8.2 was recorded on 16th March 2020 before these started and shows the App Health Overview page with a link to a video introducing Android Vitals [4], and the App Health Details page with no data.

[269]: Nieuwenhuis (2019), Apps sometimes crash (call stack at BaseExperienceActivity$2.run)

3: Expo is a very popular open source platform for making universal native apps that run on Android, iOS, and the web [3]. In that issue, several developers for different Android apps provide data from Google Play Console confirming they also receive similar crash clusters. The cause was not definitively traced or addressed by the LocalHalo development team; however, for the LocalHalo app the crashes stopped being reported once a new release (1.3.0) of the Android app was launched around 6th April 2020.

4: This appears as a mainly black rectangle in this thumbnail screenshot.
Meta-data

Meta-data is not about the app per se, but about the user and/or the user’s device, etc. Meta-data may help developers with bug localisation and reproduction pertaining to the device model, its underlying hardware characteristics, the release of the platform, and so on. An example from a discussion on StackOverflow includes an answer with code that records additional meta-data such as the available memory [270].

Figure 8.5 provides an illustration of the privacy policy for Fabric Crashlytics which lists various the meta-data it collected at the time. The successor Firebase Crashlytics lists similar data being collected for crashes https://firebase.google.com/support/privacy#crash-stored-info. The details of why these meta-data were necessary were discussed online by Mike Bonnell, one of the Crashlytics engineering team, in response to a question on StackOverflow [271].

Of note, some app developers may receive data they didn’t expect, particularly if they migrated from Fabric Crashlytics to Firebase Crashlytics.

To provide some additional context Fabric and Firebase both offered facilities to combine various datasets into their reporting, for instance based on advertising SDKs. This led to reports that included demographics in addition to the crash analytics, etc. [272].

The forced migration from Fabric Crashlytics to Firebase Crashlytics had two stages, the first was to migrate the project to the Firebase user
interface and the second was to replace the Fabric SDK with the Firebase SDK. The Firebase SDK automatically collected additional data [274].

As mentioned in Section 7.4.1. Tradeoffs, the Catrobat project chose to stop using Firebase Crashlytics when they discovered that the demographics of the end users were also being recorded.

In collaborative research into using Firebase Analytics for logging, 50 of 107 active Android opensource projects initialised just the Firebase Analytics SDK; they did not use any other aspect of the SDK [9] 6. Therefore the contents and the limitations of the default meta-data are of particular interest, since default meta-data is the only information those developers would have available to them. The remaining 57 projects used additional API calls to record additional information on one or more code-paths in the respective app.

Engineering challenges

Engineering challenges relate to developing the components of the mobile analytics tool/service such as provision of a client-side SDK that collects failures for native (C++) code.

Engineering challenges for mobile analytics include:
Support for collecting information from native code. This can be particularly pertinent for apps that include libraries in native code that are provided by third-parties.

Collecting information from the earliest stage of app startup to the app’s shutdown; otherwise data collection is incomplete, a recent example of a bug in the crash reporting is [263]. Similarly, reporting an application crashed event did not work when using Segment’s Android SDK [275].

Making the SDK reliable, performant, and virtually failure-free. Segment provide a detailed comparison of how their Swift and Kotlin achieve these compared to their older iOS and Android SDKs. Sometimes the opposite happens where the client-side SDK actually causes failures, e.g. causing ANRs [276, 277] or crashes [278].

Establishing and maintaining sufficient information to calculate and provide sufficiently accurate comparisons, ratios, and so on. As an example, determining the Probability of Failure On Demand (PFOD), requires counts of non-failures – those events/transactions/etc. that worked. Also, the sources/inputs/conditions that contributed to the failure may be useful to the app developer; does the mobile analytics SDK collect these? In the Kiwix case study;
Findings: Mobile Analytics Tools and their Artefacts

There were various sources of WebView crashes, these needed to be identified in order to attempt to prevent similar crashes in future.

- For the Vitals Scraper utility developed as part of this research, there were engineering challenges in first developing and then maintaining an automated interface to obtain reports and related information from Google Play Console and Android Vitals.

- For platform tools, collecting pertinent information across the process boundary includes engineering challenges. For in-app analytics, collecting information, such as ANRs, was a challenge during the period of the active app-centric case studies.

These challenges are ongoing, various SDKs aim to address one or more of them.

8.1.2 Developer experience

The design of the User Experience (UX) of the mobile analytics tool for their audience of the software development team (and particularly the app developers).

Vying for the attention of developers

Mobile analytics tools compete for finite attention developers are able to provide. This competition occurs at the initial selection and integration phases and continues during the life of the app. This includes gaining attention on an ongoing basis to communicate their alerts, reports, etc. Pricing, licensing, and management approvals sometimes prevent some development team members from being able to use the tools directly.

- Access and use of mobile analytics tools allows them to be used interactively, where this is impractical copies, extracts, etc. of reports and related material helps preserve them for future analysis, for evidence, and so on. These copies and extracts can also extend the reach to people who don’t have direct access to the tools. Several of the app-centric case studies, including Kiwix, Catrobat, LocalHalo, and the Commercial project, C1, limited access to various tools to a subset of the developers. In contrast, Moonpig, provided access to every member of the development team who had access to the source code of the app.

- When the analytics tools lack the attention of developers the effects of existing and new issues propagate and may enable these issues to snowball. With LocalHalo there where acute effects, described in Appendix A in Section A.2,. The Kiwix project provided a good example of this with the loss of the lead developer for the Android app at the end of 2020; and similarly for Catrobat with the loss of the Product Owner in mid-2020.

- The majority of the app-centric case studies used multiple mobile analytic tools. Some of the developers chose to ignore aspects of particular tools, for instance the crash analytics in Android Vitals in favour for similar services from other tools, even though Android Vitals is able to record some crashes that the other tools do not capture e.g. owing to limitations in the respective SDK. The tools need to convince developers of the merits of their reports. The
Moonpig development team, in particular, checked the reports of multiple tools to reduce their blind-spots.

- The LocalHalo project illustrated the flip-flop of failures between two analytics tools. This was insightful and demonstrates the value of having a combination of platform-level and in-app analytics, at least for apps written in frameworks such as React Native.

### Design of the mobile analytics events and content

Of all the mobile analytics tools covered in this research Iteratively uniquely focused on the design and verification that the mobile analytics captured the intended data. Their various software tools helped the teams design and implement the desired mobile analytics correctly in a mobile app.

**Author’s Note 8.1.1 Industrial example of a major disconnect between perception and reality**

In my industry experience, not otherwise covered in my research, I discovered a profound disconnect between what the engineering leadership said the mobile analytics collected, compared to what it actually collected in the mobile app. Over 90% of the claimed mobile analytics events were not implemented in the app. Had that project used tools such as those provided by Iteratively, the mismatch would have been identified by the software tools and clearly presented. Furthermore the client-side tooling Iteratively provided was able to fail the build of the app so the app wouldn’t have been able to be released as-is i.e. without the correct implementation of mobile analytics.

### 8.2 Fitness-for-purpose

Simply put, the mobile analytics tools need to be fit for purpose. In the context of this research, fit for purpose means that the tools need to:

- fit the needs and desires of the developers (product fit);
- provide actionable reports;
- integrate into workflows; and
- be a good return on any investment the developers make in terms of using the tools.

When the tools integrate into the workflows of the developers, they’re more likely to be adopted as long-term companions and therefore demonstrate they provide a good return on investment.

Note: two of the topics: fidelity and ethical considerations bridge both this section: fitness-for-purpose and the dependability section, they are covered in Section 8.5. Cross-cutting topics.
8.2.1 Product Fit

Product fit addresses whether, and if practical how well, the mobile analytics product fits the desires/needs of the developers and their organisation. It is similar, but more specific than product/market fit \(^7\) — or conversely the market is pico-sized, gauged at the level of a development team.

Developers of mobile analytics tools, such as Iteratively \(^8\), seek ways to identify and determine what app developers will find useful.

In interviews with Iteratively’s CEO, he explained they used various techniques including ‘ten dots’, illustrated in Figure 8.6, during one-to-one semi-structured interviews to help Iteratively prioritise the features they developed and provided. Each interviewee was given the opportunity to complete this exercise, an example of their ‘dot-voting’ \(^{279}\) is also provided in Figure 8.6. The CEO provided access to their live document containing the results of the ‘dot-voting’ and gave permission to analyse it.

Iteratively had developed a combination of an online design tool that created a schema for third-party in-app analytics tools. They also provided build tools and a small client-side SDK (i.e. a mobile SDK) which validated the schema had been implemented adequately in the mobile app. Iteratively’s SDK was intended to limit the collection to only the data contained in the schema. This was intended to stop the collection of other data (such as PII data). Through the use of the SDK, the build tools,
8.2 Fitness-for-purpose

and the online design tool development teams and their colleagues with other roles such as marketing, product, and so on could have a consistent and coherent understanding of the data that was being collected. Of interest to this research, they did not address the collection of errors or failures such as crashes in their SDK.

LocalHalo are a good example of a small development team who chose to include several mobile analytics SDKs into their mobile app where each SDK (and the respective service) was chosen to provide orthogonal data. They chose Sentry for technology facing analytics and MixPanel for product (business facing) analytics. Although Sentry provides APIs\(^9\) for integration and data forwarding and MixPanel provides data pipelines\(^10\) LocalHalo did not use these.

In contrast for the large commercial project, C1, data integration was deemed vital by the organisation in order to facilitate ongoing and ad-hoc analysis across multiple sources of information. For C1, the product (the mobile analytics tool) also had to fit at the level of the engineering organisation where the mobile analytics needed to be ingested into the corporation’s ‘data-lake’.

Finally for this topic, despite the many mobile analytics SDKs, there may be situations where none provides the answers the developers seek. As a concrete example, SmartNavi uses Firebase and Google Analytics, mainly for tracking the popularity and the use of the app’s features. The app also incorporated Fabric Crashlytics for crash reporting. Nonetheless the developer explained none of these analytics products provided analytics related to software running in the background, as a background process on Android. SmartNavi provides GPS services to other apps and runs in the background. As the Android Platform has evolved Google has embedded restrictions that limit and constrain background processes which meant the SmartNavi software is suspended (paused) by Android. The developer would like to improve the app’s behaviours when it runs in the background but lacks the analytics to do so.

8.2.2 Actionable Reports

Actionable Reports are reports the developers can use to decide on what should be done to address concerns presented in the reports.

Several of the app-centric case studies materialised because the respective development teams became aware of excessive and chronic error rates for their mobile apps. These projects (Kiwix, Catrobat and C1) had not managed to materially reduce the high crash rates directly and they were happy to receive help and insight in how to apply the information from the reports in their work. At the time they lacked the wherewithal to do so unaided. With interventions, including those that were part of this research, each of the development teams were able to materially improve the error rates of their respective apps.

None of the currently available mobile analytics tools seen during this research were able to pinpoint the causes of failures, instead they identify one or more effects e.g. the app crashes, or is stopped by Android because it became unresponsive. The action-ability comes in part from the metadata collected by the various mobile analytics tools which helped in bug
localisation and in part from the characteristics and patterns contained in the reports. The Moonpig developers, for example, were sometimes able to identify a highly likely reason for the failure from the contents in the reports. They took action by modifying the application’s source code with the aim of addressing the problem reported in the mobile analytics service.

Android Vitals aims to highlight emerging problems with a deployed app. For instance if there is an acute and significant increase in the crash rate for the production release(s). It also provides cross-connections between problems found in Google Play’s pre-launch reports and failures that occur in production. Developers found some of the reported failures easy to comprehend and address, others were less tractable. Several of the developers who use Crashlytics said they preferred using it over Android Vitals for comprehending crashes. Crashlytics provided more information than Android Vitals and the reports were easier to digest therefore the Crashlytics reports were more actionable than those from Android Vitals.

As discussed in the flaws topic (Section 8.4.1. Flaws in the mobile analytics tools and/or services) various reports have flaws, for instance in their aggregation. There is scope to improve the mobile analytics tools through improving the matching process in the data aggregation (for instance, where there are fragmented Failure clusters) and in the analysis across multiple legitimate failure clusters (such as across all NullPointerException Crash clusters) to help identify underlying flaws in the development of the app.

The Moonpig case study provides an illustrative example of how the development team was able to take proportionate and measured action as the crash-rate of their Android app increased. The cause was related to a known and documented issue with a third-party software library, RoboSpice. There was a clear correlation where the crash-rate increased on newer Android releases. The team was able to evaluate and estimate an appropriate timescale to replace that library by revising the application’s source code. They were able to schedule the release to suit other strategic objectives rather than rushing to push out a ‘fix’ of the app.

The release management reports in Google Play Console (that incorporate various Android Vitals reports pertinent to the latest release for the 7 days post release) were highly actionable for the C1 project. The development team were able to abort releases that had unexpected increases in failure rates before those flawed releases adversely affected swathes of the userbase.

### 8.2.3 Integration into workflows

Integration into workflows is the ability of a given mobile analytics tool/service to be integrated into development team’s workflows. These workflows include bug tracking and release management. Some mobile analytics tools, e.g. Microsoft App Center, provide facilities where developers can mark and/or annotate elements in the reports, for instance to remove a crash cluster from the main report or to provide a bug-tracking link to help the developers streamline their work when using mobile analytics.
When cross-references are stored in mobile analytics, viewers can see which of the issues have been reported (and by inference which have not been acknowledged e.g. because the issues are new to the team).

Figure 8.7 illustrates various aspects of integration between mobile analytics and other software tools such as bug trackers. The mobile analytics tool provides reports in a graphical user interface. On the web, the report has a URL (it may not be directly addressable in a mobile app equivalent tool) and visiting the URL with a suitably authenticated account results in the report being rendered if the mobile analytics service is working adequately. The mobile analytics tool may provide interactive elements in the report page, for instance to hide the issue, to annotate the report, to modify the selection criteria.

Some tools, e.g. Android Vitals, generate URLs that include structural elements for instance to vary the dates or the duration of the report (24 hours, 7 days, etc.). And some tools provide for easy export of elements of the report or of the entire presentation of the report e.g. as a PDF file or an image file. Another approach, albeit one seldom supported officially is to write code to extract content programmatically; we did so as part of this research and developed Vitals Scraper to extract content from Android Vitals.

**Why do these aspects matter:** Development teams often want to identify
and track bugs including any code changes made with the aim of fixing the bug. Mobile analytics may be the source for the information about the bug and when the issue in the bug database includes pertinent information from the mobile analytics tool the development team can use that information to help understand the bug. The ability to cross-link the mobile analytics report and the issue in the bug tracker can enable team members to review the current, live information in the peer system.

Figure 8.7 includes five distinct mechanisms to integrate individual reports, or at least some of their contents.

1. The web address or URL: for some reports the URL may include structural elements that affect, or vary, the contents of the report.
2. Exports of the report: these are generally as a PDF or as a Comma separated values (CSV) file (which would include a subset of the entire report).
3. Ubiquitous screenshots: some web browsers e.g. Google Chrome limit the screenshot to the viewport (what’s visible in the web browser), others e.g. Firefox provide options to take a screenshot of the entire report.
4. Content extracts: e.g. using copy+paste links on screen, e.g. Android Vitals does so for the stack trace of crashes and for the thread dump of ANRs.
5. Programmatic extraction via the GUI: commonly known as screen-scraping or web-scraping.

Text-based content can be stored and processed relatively easily (automated analysis and processing of graphically-based PDF reports and images is beyond the scope of this research).

Report exports: Android Vitals provides various exports as files of comma-separated values (CSV files). However, these are encoded in a UTF-16 character encoding, an encoding used by less than 0.005% of websites according to [280], via [281]. UTF-16 content proved awkward to parse programmatically which complicates and adds friction to any analysis or integration of these CSV reports from Android Vitals.

What content does a URL point to? Where the URL points to can affect the integration where a) the underlying content has been updated and the reference was intended to be of a snapshot in time when an issue occurred, b) the underlying content has not been updated yet the url indicates the view should be relative to the current time and date, or c) where the content is no longer available, at all.

The URLs may point to persistent historical reports or to recent, periodically updated, reports. Some reports are also updated as new releases of an app go into production. The URLs and/or the contents often have a finite life. Trend analysis might be available within the mobile analytics tool, otherwise it can be performed using snapshots of reports for a range/set of consistent periods e.g. snapshots taken on the first day of every week can be used to perform trend analysis. Retention policies applied by the mobile analytics service may mean the underlying data is deleted; for example Firebase Crashlytics only retains crash stack traces, etc. for 90 days [282]. The data is then deleted from Firebase Crashlytics. Therefore, development teams would need to preserve the contents of

---

12: Note: some mobile analytics tools may provide API access to obtain the reports and/or the contents of the reports however none was discovered during the case studies.

[280]: Q-Success trading as W3Techs (2022), Usage statistics of UTF-16 for websites
[282]: Google Developers (2022), Privacy and Security in Firebase
the reports if they wish to perform longer-term analysis or for auditing, etc.

**Integration of event and content verification:** As mentioned earlier in this chapter, Iteratively’s various software tools helped the teams design and implement the desired mobile analytics correctly in a mobile app. Their tools were primarily involved before the app was tested or released, nonetheless they included a short duration view of the mobile analytics events being emitted by the app when the app was being used. In this research they were used in the IDoT micro experiment.

In comparison, the release management section of Google Play Console is integrated into the rest of the services and reports offered in Google Play Console. Therefore it knows the details of each release and is able to perform calculations and analysis accordingly for every new release of an app. In the CI project the product owners were also able to use the release management reports to help them work with the developers to make decisions on staged rollouts including, when necessary, stopping the rollout entirely. The release management tools were a core part of release management and rollout for the overall team.

The final topic for the integration into workflows is on pipeline integration between mobile analytics tools/services and other high-volume data processing systems used particularly in larger tech-savvy enterprises. In the corporate CI app-centric case study the organisation required the product’s development team to integrate any and all of the mobile analytics services where such integration was available. These were paid-for integrations where the service provider charges for the integration and for the data processing aspects. The app developers did not necessarily use the internal data lake often, instead the Operations and Site Reliability teams did, therefore the details are beyond the scope of this research.

### 8.2.4 Programmatic access to analytics outputs

None of the mobile analytics tools encountered during the research provided complete access to the outputs using APIs. Furthermore, the ability to record and preserve copies of visual reports (as well as the underlying data) facilitates both practical use of the data in the field (for instance to record the information in an issues tracking database) and for further analysis and research.

In 2013, API access to mobile analytics was requested for Google Play’s Developer Console [283] and various people have developed code that interfaces with Google Play Console in an attempt to provide automated, scripted access to the content. In March 2022, Google launched API access to Android Vitals [284]. According to the documentation it remains a v1 alpha release ¹³ and currently provides various summary statistics, it does not include any details of failure clusters. It was launched after the app-centric case studies so it was not practical to determine if any of these development teams use(d) it.

---

[283]: Fecherolle (2013), *Getting statistics from Google Play Developers with an API*

[284]: Myton (2022), *Access Android vitals data through the new Play Developer Reporting API*

¹³: developers.google.com/play/developer/reporting
Two examples of opensource projects that were created to download data include:

googleplay_dev_scraper\textsuperscript{14}, provided mechanisms to automate the downloading of the CSV monthly reports rather than the live reports. It was last updated in 2013 so no longer current.

There was also the Andlytics opensource app that provided developers with access to data from their Google Play Console account\textsuperscript{15}. However this project also ceased active development for various reasons, probably because Google chose to restrict access to the underlying data in 2019\textsuperscript{16}.

Using Web-scraping for ad-hoc integration: Web-scraping of content from web-sites continues to be a frequent activity performed by many people and services. web-scraping is used in many fields including bioinformatics, where the authors discussed why web-scraping was still necessary in a world full of API’s\textsuperscript{285}. A more recent paper by\textsuperscript{286} briefly presents various inefficiencies in web-scraping. Curiously this paper singles out journalism as an under-served area despite it being written about 7 years earlier in Chapter 4 of\textsuperscript{287} (and also covered in a more recent version of the book,\textsuperscript{288} pp. 133, 238). Suffice to say, web-scraping is a topic that has been written about, albeit not in the context of scraping content from mobile analytics web interfaces.

We developed Vitals Scraper as an opensource project\textsuperscript{289} and released it as an npm package\textsuperscript{290} to address the issue of the lack of access to the outputs of Google Play Console and Android Vitals. Doing so demonstrated the necessity, viability, and some of the maintenance challenges of writing automated software for web-scraping of outputs from Google Play Console with Android Vitals.

Data Export integration: enterprise-grade mobile analytics services provide mechanisms to export data to homogeneous data storage platforms\textsuperscript{213}; for instance Google Analytics tools export content to Google Cloud Storage\textsuperscript{17}, and provide mechanisms to automate the process\textsuperscript{18}.

API Access: Microsoft App Center is unusually complete and provides a comprehensive set of open APIs openapi.appcenter.ms/ as well as the ability to export analytics data to Azure docs.microsoft.com/en-us/appcenter/analytics/export and even export data for individual users docs.microsoft.com/en-us/appcenter/gdpr/analytics-export. Google Analytics provides various reporting APIs including those for Exceptions\textsuperscript{19} which are collected by Firebase Analytics for both Android and iOS apps\textsuperscript{20}.

Some of the mobile analytics providers offer developers customisation options such as custom dashboards and reports in Sentry docs.sentry.io/product/discover-queries/uncover-trends. As a broader observation, various companies provide app analytics services where they obtain the underlying data and aim to provide easy to use, attractive, and actionable reports; examples include appfigures.com and www.data.ai/en (previously known as AppAnnie). Appfigures also

\begin{itemize}
\item \textsuperscript{14} github.com/tmurakam/googleplay_dev_scraper
\item \textsuperscript{15} github.com/AndlyticsProject/andlytics
\item \textsuperscript{16} github.com/AndlyticsProject/andlytics/issues/766
\item [285]: Glez-Peña et al. (2014), ‘Web scraping technologies in an API world’
\item [286]: Diouf et al. (2019), ‘Web Scraping: State-of-the-Art and Areas of Application’
\item [287]: Gray et al. (2012), The data journalism handbook: How journalists can use data to improve the news
\item [288]: Bounegru et al. (2021), The Data Journalism Handbook: Towards A Critical Data Practice
\item [289]: Harty et al. (2019), Vitals Scraper source code on GitHub
\item [290]: Harty et al. (2019), Vitals Scraper npm package
\item [213]: Mayot (2015), Integrate Play data into your workflow with data exports
\item [17]: support.google.com/googleplay/android-developer/answer/6135870
\item [18]: cloud.google.com/bigquery-transfer/docs/play-transfer
\item [19]: ga-dev-tools.web.app/dimensions-metrics-explorer/exceptions
\item [20]: developers.google.com/analytics/devguides/collection/firebase/android
\end{itemize}
8.2 Fitness-for-purpose

publishes status reports for the performance of Google Play and Apple’s App Store Connect service, these track the publishing performance of these two app stores; at the end of February 2022 they observed Google has been several days late publishing their free daily reports.

8.2.5 Android Vitals

As Android Vitals featured in every one of the app-centric case studies it has been possible to consider it in greater depth than the other mobile analytics tools investigated in this research. For example, having access to Android Vitals enabled comparisons across case studies and across multiple apps in and across those case studies. Having access also led to direct investigations of the Android Vitals on all the app-centric case studies. Flaws could be compared, for instance: 1) where the second country in the userbase graph was mistakenly used for the title of that report, 2) where a userbase appears to be negative when more users were lost and gained over the lifetime of the app, as shown in Figure 8.8. This has been observed for several of the apps in the case studies, a pattern for when this occurs has yet to be determined.

Note: Appendix A also includes two worked examples: Section A.4 for the Dashboard report, and Section A.5 about a period where it was unclear whether Android Vitals was actually processing and/or reporting on the underlying data.

Which apps can Android Vitals help?

As Google only provides various reports in Android Vitals once they decide enough data exists to preserve the privacy of end users Android Vitals provides little for developers of less popular apps.

Based on very rough approximations combining my case studies with AppBrain’s download statistics to 19th June 2019, shown in figure 8.9[291], of the total populations of app developers:

- ▶ 3% to 4% (those with 100,000 to 500,000 total downloads) will get limited value as at least one report will be provided.
- ▶ < 1% (those with 500,000 to 1,000,000 total downloads) will get some value as many of the reports will be provided, but not all.
- ▶ 1% (those with > 1,000,000 total downloads) will get extensive value as most/all the reports will be provided.

Limitations in using Android Vitals for Observability

Observability provides two key benefits according to [292]: 1) monitor key signals, and 2) understand changes in a system. Mobile analytics facilitates observability of mobile apps, and in particular Android Vitals facilitates the observation of app startup times, performance, ANRs and crashes that occur when an app starts-up. However the inability of storing and analysing the data over time in Android Vitals (beyond the standard reports and recent failure details) limits the ability to observe and analyse stats and/or failures over time.

[292]: Sigelman (2021), Observability will never replace Monitoring (because it shouldn’t)
198 8 Findings: Mobile Analytics Tools and their Artefacts

Figure 8.8: Google Play Console Dashboard for PocketCode Lifetime User KPIs, on 24th Jan 2020

Figure 8.9: AppBrain: Download distribution of Android apps, from June 2019
Who gets sufficient usage to see more detailed reports?

Our investigation of Android Vitals indicates that reports are only provided when there is sufficient data collected to ‘prime the pump’. It may be possible to estimate how many apps of those in Google Play Store are likely to have enough volumes of usage data. Google makes various recommendations for developers on how to apply the results Android Vitals reports developer.android.com/distribute/best-practices/develop/android-vitals however the developers can’t do much until Android Vitals actually shows them the data! Mainstream apps with less than about 50k active installs \(^{21}\) are unlikely to receive comprehensive detailed reports.

- These ‘active installs’ counts are around 20% to 30% of the total installed user base for various apps used in our research e.g. the active installs would be around 20k for an app that shows at having 100,000+ [total] installs to end users in Google Play.
- Apps with atypical characteristics, whether they are much more unreliable than their peers, or used on a small number of device models, and so on may receive detailed reports as those characteristics would mean Google Play’s internal thresholds would be met sooner. An example of this occurred when the Expo issue occurred and suddenly the Android Vitals crash cluster reports were available (until the crash rate decreased after the issue and submerged under the threshold again when the reports stopped).

Data provided by AppBrain [291] was used to estimate the populations of apps that are not likely to generate enough data to see various reports in Android Vitals. Based on Android Vitals reports for Kiwix custom apps we infer that few apps with less than 20,000 total installs will have any detailed reports; WikiMed in Spanish has 5,373 active installs and had one report, for crash rate by app version. None of the other reports are available for this app. As the CTO of Moodspace observes on Page 8.3.1 Android Vitals did not provide him with the crash details even though their app had 4K monthly users.

The threshold for when there is enough data for Google to provide a report depends on various factors, so the total installs is a proxy measurement and imperfect. Therefore as a rough estimate Android Vitals is unlikely to offer much value for developers of (973,381 + 730,419 + 553,261 + 284,634) apps i.e. 2,541,695 apps in Google Play, those with less than 10,000 downloads. For the next 92,678 (those with 10,000+ downloads) the value of Android Vitals might increase somewhat, depending on how their app behaves and their user-base (e.g. are they on a few Android versions or spread across a spectrum - the larger the spread the less likely the reports will have data). And so on. By the admittedly limited view into the overall data set, Android Vitals is best placed to help the developers of the top (21,728 + 35,854) 57,582 apps, approximately 2% to 3% of the total population (2,691,955 apps). These apps (according to AppBrain’s data on library use) are also more likely to use Firebase, Crashlytics, etc. so also have some of the run-time data available from these sources in addition to Android Vitals.

This research investigates two broad sources of data - data collected by the operating system (here effectively what appears in Android Vitals)
and data collected using in-app libraries, particularly mobile analytics, it could include heatmapping (e.g. Appsee, found in over 790 Android apps with over 375 million downloads [293]), crash handlers, etc. to provide feedback to measure how well the development team did in terms of testing and code quality. What they learn could also be useful to help them improve how they develop and test their apps in future, particularly with the greater detail mobile analytics (particularly Firebase) can provide the team.

False positives

Not all issues flagged by Android Vitals are valid problems that need to be acted on for a given app. For example, in the Kiwix app-centric case study, Android Vitals sometimes reports excessive network usage running in the background while the device is running on battery, as shown in figure 8.10. However, as the Kiwix app was designed to enable users to download sometimes extremely large files, and to do so in the background, this warning is to be expected and not a bug - it’s a feature. At the time of the case study the app downloaded the catalogue of content available to download shortly after it starts for the first time, exactly the conditions reflected during the pre-launch testing.

App developers could choose to modify their app’s behaviour to increase their score in Android Vitals; the effects could actually improve the app’s behaviour for end users, or potentially the effects are short-lived and purely to defeat the automated adverse assessment – in a similar fashion as some vehicle manufacturers who implemented ‘anti-defeat’ devices to improve the measured results of their vehicles during authorised testing [294, 295].

8.3 Utility

Utility addresses practical and pragmatic aspects of developers using mobile analytics to achieve their purposes. It’s a close cousin to Section 8.2.
8.3 Utility

Fitness-for-purpose and considers three discrete lower-level themes: efficacy of the mobile analytics tool(s), the benefits of combining tools, and bug-localisation.

An orthogonal topic is included in this section as Google has provided various online tools to help developers pre-release of their Android apps. One in particular, pre-launch reports, compares failures found pre- and post-release and is instructive in how well automated platform-provided checks and tests might help developers reduce the incidence and/or severity of post-release failures.

8.3.1 Efficacy of the tool

Efficacy of tools considers how efficient and how effective is the tool, i.e. how efficacious is the tool? Did it achieve the objectives it claims to achieve?

**Android Vitals:** The current design of Android Vitals includes notifications and alerts, the use of colour, comparisons, and so on are all intended to get the attention of the developers and help the developers focus on egregious issues. During the app-centric case studies several project teams provided examples of where these worked. Comparisons with peer apps helped inform and motivate the project team for the commercial app, C1, albeit the product owners appeared to be more interested than the core development team. (Note: the comparison is provided per developer account, per app, rather than per logged-in user, in Google Play Console, i.e. anyone able to login to see the app’s analytics will see the same set of peer apps.).

> “Seeing your app in comparison to apps of peers provides some great motivation to step up your game.”, Moodspace (2019).

> “...we noticed them by monitoring with Crashlytics/Firebase, I think specifically in the last case we got a crash velocity alert”, Moonpig (2020).

For the Kiwix project, the project leads were convinced by the reports provided in Android Vitals that an interim bug-fix release would be worth the effort of making and releasing quickly in order to materially reduce the crash rate for the core app (it was - as measured by Android Vitals). Note: this is covered in greater detail in Appendix A in Section A.1.

**Crashlytics and Microsoft App Center:** both include in-app crash reporting that project teams preferred using over the equivalent reports in Android Vitals. The reports were more actionable for at least two main reasons, one systemic and the other because of how Google currently processes and reports on crashes in Android Vitals.

The systemic reason is that in-app SDKs often offer facilities that enable developers to get the apps to report information in addition to crashes. The information may include: breadcrumbs, custom data, handled exceptions (non-fatal errors), and so on. As Android Vitals relies on data that is logged when an app is terminated it does not have access to any of these additional (and optional) data.
Android Vitals limits what it reports to the app developers. Here are two key examples.

1. Android Vitals currently doesn’t present the optional message parameter, e.g., the parameter ‘s’ in Java’s `NullPointerException(String s)` \(^*\). While some crashes can be determined without this message (and indeed some fatal exceptions don’t include a message as it’s an optional parameter) others have only been determined by the developers when the developers have seen the contents of the messages.

2. Android Vitals also only provides reports once internal thresholds have been reached whereas the in-app analytics tools report even isolated events. With Android Vitals Google has chosen to protect user’s privacy (possibly relatively rather than absolutely) compared to their other mobile analytics product offerings such as Firebase Analytics and Firebase Crashlytics.

This second example was raised as a requested improvement during the interview with the CTO of Moodspace:

“The only issue I have with core vitals is that I can’t see them all! We are by no means a big app, so don’t have enough data to meet Google’s standard for anonymised results, so results for most of the core vitals are hidden. I don’t quite understand why this should be the case as the headline figure of your apps performance surely doesn’t have to rely on anonymous data? Whereas the drilled down details of a core vital should be anonymous, so maybe the details view could just be blocked instead of hiding the entire core vital? To provide context, MoodSpace had at least 4k monthly users, so there must be plenty of apps which get little or no use form core vitals, simply from them being hidden.” Moodspace (2019).

Auto-instrumentation, for example, provided by Sentry and as seen in the LocalHalo project, can also help developers to identify contributory factors to failures. In the LocalHalo case study, Sentry clearly identified the servers were no longer functional. This occurred months after the project and the app were no longer supported, were the project actively maintained there was sufficient information in the analytics to help the development team focus their attention on the servers. In the commercial project C1 their mobile analytics services did not include equivalent auto-instrumentation, instead one of the developers wrote similar code to log (i.e., record) web server responses. In doing so they inadvertently caused the newly released version of the Android app to crash frequently. Here what is noteworthy is that developers sometimes have to write their own code to augment the functionality provided by a mobile analytics service. It’s pertinent to mention that auto-instrumentation can also have flaws that include causing the app to crash. A similar example, albeit for a different third-party SDK was when Facebook’s SDK led to high app crash rates.

\(^*\) In full, `NullPointerException(String s)` Constructs a `NullPointerException` with the specified detail message. Source: docs.oracle.com/javase/7/docs/api/java/lang/NullPointerException.html
8.3.2 Benefits of combining tools

Moonpig’s development team actively combined Firebase Crashlytics and Google Analytics for diagnostics in addition to the information available in Android Vitals. They estimated they used Android Vitals approximately 30% of their time to identify flaws and issues related to their Android app compared to using the in-app analytics. They also incorporated in-app analytics by using Firebase Analytics which recorded analytics related to how the users use the mobile apps and whether errors or other problems occurred while the app was being used.

Some crashes were only reported in Android Vitals, Moonpig’s hypothesis was that as Crashlytics waits to send crash reports until the app next restarts, some users never restarted the app therefore those crashes remained unreported even if crashlytics had detected and recorded them.

subsection Bug-localisation

Moonpig provided several examples of how mobile analytics helped with bug localisation. The most instructive example is where there was a clear correlation between an increase in the crash rate of the app and newer Android releases. The development team:

(a) understood the correlation, in part as the early effects were documented by the developer of the relevant third-party software library: RoboSpice, and

(b) they were able to make suitable purposeful, controlled improvements to the app where they were able to revise the codebase so the app could be improved despite the removal of RoboSpice.

With Kiwix, Android Vitals provided sufficient information to help the developers to gauge the probable impact of various flaws that emerged in use of the family of Android apps. The reports weren’t necessarily comprehensive or complete so they didn’t necessarily pinpoint something the developers could address. Similarly for the SmartNavi Android app the developer was able to determine the crashes that wouldn’t be fixed because the analytics reports indicated the failures were a) sufficiently sporadic and infrequent, and b) on devices that weren’t available outside their respective region e.g. China. So the reports helped the developer of SmartNavi to make appropriate triage decisions - the action was to take no further action!

8.3.3 Pre-launch reports

Pre-launch reports are an intrinsic part of Google Play Console and the pre-launch report includes automated testing of pre-release apps. These reports are generated by a combination of static analysis tools and automated ‘testing’ performed by a software test facility called Robo, which is also used in the Firebase Test Lab service.

Pertinently, Google Play Console identifies issues that are found in both the pre-launch testing and in production, an example is provided in Section 6.4.3. Pre-launch testing. They also automatically select the top five languages based on the app’s userbase [296], which is a simple example of how mobile analytics can be used to drive software testing. This finding illustrates, albeit at a basic level, a topic raised in Section 32.

---

22: Google provides a current overview of pre-launch reports online play.google.com/console/about/pre-launch/reports.

23: More probably they actually select the locales which are more specific than simply a language.

[296]: Google (2022), Use a pre-launch report to identify issues.
Prioritising devices to test on where the suggestion of prioritising testing based on mobile analytics was mooted.

“If you use pre-launch reports to identify issues with your apps, crashes found during testing are listed with your app’s crashes and ANRs. However, because crashes found while generating a pre-launch report come from test devices, they don’t affect your crash statistics.” [297, 298].

Figures 8.11 and 8.12 contain examples of crash clusters that were detected in production that were also found by pre-launch reports through the pre-launch automated testing. The first of these figures is the more granular and more detailed report, the second figure contains two extracts from the ordered columnar reports of crash clusters for the app. Collectively they demonstrate that the pre-launch reports can, and sometimes do, provide developers with early warning of crashes before the app was released into the wild. The data collection, analysis, and findings are described in Appendix A in Section A.3.

As the GTAF project noted, the crashes reported in pre-launch reports
do not necessarily affect end users. Conversely the pre-launch report automated testing does not find all the failures that affect end users. (Dua & Zikr app).

Unfortunately the pre-launch reports started reporting false positives claiming that apps were crashing during the automated testing; and this was exacerbated by the tardy responses by the Google engineering team and inadequate ‘fixes’ that didn’t work [299]. Some app developers chose to bypass the pre-launch reports so they could release new versions of their apps. These false positives adversely affected several of the app-centric case studies including the commercial project CI that had experienced other flaws in the pre-release release tracks serving outdated releases to key stakeholders. The pre-launch report for the WikiMed app in Arabic also reported this issue, see Listing 4. The crash appears to be in the unrelated YouTube app installed on that device.

Listings 5 and 6 are extracts of stack traces reported in pre-launch reports in 2018 (30+ lines of non app-specific stack traces are removed from both these extracts to improve readability). They were reported, together with crashes that occurred elsewhere in the app, during the period (2018 - 2019) of the excessively high crash rates where the custom file downloading code was quite buggy.

Three lines of four lines for the Kiwix codebase match in the stack traces, the fourth in each is specific to the action - to play/pause the download of a file. The same exception occurs in both cases, an `IndexOutOfBoundsException` because there are zero elements in an internal array memory structure in the app. The cause of these crashes was likely to be in-common, however as this functionality was stripped out of the codebase shortly afterwards the investigation is beyond the scope of this research. In short the automated testing helped identify a bug probably common to both code paths. The development team had not identified the bug previously and the pre-launch’s Robo testing service identified two of the crashes that were probably affecting the userbase 24.

Empirically observed, pre-launch reports are preserved for several years (e.g. for 6+ years for 30 releases of the Physics Education Technology. At their request the Kiwix project renamed our custom app from PhET to ‘Physics and Chemistry simulations’. In this research PhET is commonly used for brevity. (PhET) Kiwix app from 19th June 2016 for release 2, to 4th April 2020 for the current release 6200950, and for up to 100 internal releases (e.g. Kiwix) from 19th June 2019 for release 4220501, to 30th Jan 2022 for release 7230406.

For the two main Catrobat apps there are 28 reports for Pocket Code starting from 8th March 2018 for release 56, to 16th May 2021 for release 86; and 18 pre-launch reports for Pocket Paint, from 13th May 2021 for release 39, to 3rd June 2022 for release 48.

In summary, pre-launch reports have found various bugs in several of the app-centric case studies. For example they found a `NullPointerException`
8 Findings: Mobile Analytics Tools and their Artefacts

Listing 5: Extract of pre-launch crash report A for Kiwix Android app, in 2018

Listing 6: Extract of pre-launch crash report B for Kiwix Android app, in 2018

...in Locale.getLanguage() in the oldest pre-launch report for the PhET app. Their analysis integrated into Android Vitals and used production analytics to help drive the automated testing of new releases of the app. Nonetheless, pre-launch reports have their own flaws and some development teams chose not to use them as they were perceived as troublesome.

8.4 Dependability

Dependability in the context of this research considers the extent to which developers can rely on a mobile-analytics service (and/or the underlying tool; and even a tool with few flaws cannot compensate for a heavily flawed service). In the author’s professional experience, mobile analytics is inherently something that’s ongoing rather than a one-off/occasional event; therefore the key concern is in the provision of a service using an underlying mobile analytics tool. Flaws in the tool are unlikely to be improved by the service that’s provided (either in-house by the development team’s organisation or by an external provider).

During this research four lower-level themes emerged that primarily underpin dependability: 1) flaws, 2) link rot, 3) testability, and 4) trustworthiness; these are covered in this section. Two more lower-level themes in particular also contribute to dependability: fidelity and ethical considerations, these are covered in Section 8.5. Cross-cutting topics.

8.4.1 Flaws in the mobile analytics tools and/or services

Software has flaws \(^{25}\), mobile analytics tools are written in software, therefore mobile analytics tools have flaws. Various flaws have been identified in mobile analytics services, how much do these flaws matter?

As the common tool across all the app-centric case studies Android Vitals provided a backbone for analysis, and a basis for scoping, comparing, and assessing some of the key issues.

25: As freely acknowledged by Donald Knuth, see [300]

[301]: Google Inc. (2022), Troubleshoot app statistics problems
Published flaws in Android Vitals: Since 2011, Google has published a list of various changes and corrections to Google Play Console [301]. This research found numerous additional flaws, and many of these were reported to Google’s engineering team for Google Play Console and Android Vitals. Of the flaws I reported to Google, some were confirmed as being accepted; the rest remained unconfirmed (but nonetheless Google may have accepted them internally).

Ubiquitous errors in the GUI also occur

| Published flaws in Android Vitals and Google Play Console to date as part of this research. Of these, fourteen have already been published [2, 3, 303]; they are repeated in Table 8.1 for ease of reference. The 15th item was discovered as part of this research, where Google Play Console displayed newer dates than the actual release date for various apps; and the 16th and 17th items during writing up this research. This 16th item, where identical crash clusters appeared several times in the listing is not the first instance of flaws in app store related listings, for example Apple’s rankings for iOS went haywire for several days according to [304].

Seventeen flaws have been found in Android Vitals and Google Play Console to date as part of this research. Of these, fourteen have already been published [2, 3, 303]; they are repeated in Table 8.1 for ease of reference. The 15th item was discovered as part of this research, where Google Play Console displayed newer dates than the actual release date for various apps; and the 16th and 17th items during writing up this research. This 16th item, where identical crash clusters appeared several times in the listing is not the first instance of flaws in app store related listings, for example Apple’s rankings for iOS went haywire for several days according to [304].

Of these flaws, the most pivotal from a research perspective is the first one, that testing is discouraged. Google appears unwilling to subject its service to external scrutiny, and its dominance in the online world puts investigators at risk of being barred from using various services considered essential to varying degrees. In contrast, several providers of mobile analytics tools make at least some of the relevant software available under an opensource license (e.g., PostHog, Sentry, Segment, Count.ly provide both client and server code; Firebase Analytics and Microsoft App Center make client SDKs available).

Google has made numerous modifications and enhancements to Android Vitals during the period of this research and some of these changes have addressed some of the flaws that were reported to them previously. Nonetheless Android Vitals still has flaws at the time of writing, for instance some failure clusters are still fragmented into several groups rather than being fully aggregated.

Other mobile analytics tools from this research have also had flaws, for example Microsoft App Center’s reports include errors and crashes that ‘happened’ hours or even as prematurely as several days in the future. This is the eighteenth type of flaw identified in this research. And Azetone’s dashboard included several non-zero values, including 20 of 50,000 users even though the software had not been used at this point. The Azetone flaws can be grouped into a nineteenth flaw: zero should be zero.
### Table 8.1: Flaws discovered in Google Play Console with Android Vitals

<table>
<thead>
<tr>
<th>ID</th>
<th>Flaw</th>
<th>Consequences</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>Testing discouraged</td>
<td>Limits investigation into behaviours and their impacts.</td>
</tr>
<tr>
<td>02</td>
<td>Negative populations</td>
<td>Nonsensical and therefore untrustworthy statistics</td>
</tr>
<tr>
<td>03</td>
<td>Repeated graphs</td>
<td>Poor UX, waste of space (waste of real estate).</td>
</tr>
<tr>
<td>04</td>
<td>Gaps in the data</td>
<td>‘Flying blind’, loss of confidence in the service.</td>
</tr>
<tr>
<td>05</td>
<td>Inconsistent data ranges for some reports</td>
<td>Poor UX, confusing, may lead to incorrect/flawed decisions.</td>
</tr>
<tr>
<td>06</td>
<td>Missing URL parameters</td>
<td>Results incorrectly filtered.</td>
</tr>
<tr>
<td>07</td>
<td>No updates for 10+ days</td>
<td>‘Flying blind’, loss of confidence in the service.</td>
</tr>
<tr>
<td>08</td>
<td>Incorrect ranges in reports</td>
<td>Off-by-one errors.</td>
</tr>
<tr>
<td>09</td>
<td>Unexplained negative headline rate</td>
<td>Exemplified in Section 8.2.5. Android Vitals the combination of new users acquired and users lost indicates a net loss in the userbase, contradicted by other data in the mobile analytics service.</td>
</tr>
<tr>
<td>10</td>
<td>Poor grouping of clusters</td>
<td>Inaccurate summaries, rank of failures skewed, sub-optimal prioritisation.</td>
</tr>
<tr>
<td>11</td>
<td>No Service problem-reporting</td>
<td>Lack of transparency of historical service outages, etc.</td>
</tr>
<tr>
<td>12</td>
<td>Lack of reports (despite usage)</td>
<td>Unusable analytics for low to mid range usage by end users, ‘Flying blind’ after take-off.</td>
</tr>
<tr>
<td>13</td>
<td>Second country’s data conflated with that of the first</td>
<td>Misleading report, poor UX.</td>
</tr>
<tr>
<td>14</td>
<td>10x mismatch with crashlytics</td>
<td>Lack of trust in at least one of the mobile analytics tools.</td>
</tr>
<tr>
<td>15</td>
<td>Incorrect date for last update</td>
<td>Misleading developer experience, loss of trust in the service.</td>
</tr>
<tr>
<td>16</td>
<td>Several identical crash clusters in the paged list of ranked results, e.g. Kiwix issue 2482.</td>
<td>Adversely affects counts of matching crash clusters, confusing.</td>
</tr>
<tr>
<td>17</td>
<td>Stale data in some graphs</td>
<td>Data should be fresh and recent in each of the reports for the ‘last’ so many days e.g. ‘Last 7 days’.</td>
</tr>
</tbody>
</table>

**Figure 8.13:** Azetone analytics dashboard for a new account, for Kiwix in 2015
8.4.2 Link rot

This section covers two topics: link rot and preservation of results: the validity of a URL may be finite, as may the contents be even if the link remains.

For mobile analytics services link rot is often a common reason why results are no longer available from the mobile analytics service - the link was ephemeral. In such cases the results would need to be preserved while the results are still available. In some cases the rot may be easy to predict, for instance as data ages beyond the predefined date range of a report, in other cases less so, for instance when the active release is updated the data for the previous currently active release might ‘disappear’ from some reports. The effects of link rot, were endemic across the various mobile analytic tools investigated during this research. They were one of the reasons that screenshots or other copies of report data were recorded by some projects, and similarly they were recorded to support this research.

8.4.3 Testability

Testability for the purpose of this research is the ability to test the overall service, the tool, and/or components that comprise the analytics tool.

Testability applies to any mobile analytics service, including related SDK(s), the topic is wide-ranging. One area of focus is on data, including the data preservation, security, and transmission, to the data aggregation, processing, and analysis, and to the reporting. It could also apply to the human elements such as their interpretation, comprehension, and so on. Access to the underlying source code, to the design objectives/requirements, to any existing tests, etc., may help improve the testability. Conversely, commercial and legal constraints and considerations might limit and/or bias the testability of a given mobile analytics tool/service.

It’s a big subject and perhaps far larger than the scope of many PhDs, certainly it is beyond the scope of this PhD which focuses on other aspects. For the purposes of this research testability focuses on working within commercial constraints and on assessing the testability aspects of testing any of the mobile analytics products/services. In earlier topics, in Section 8.2.2 and Section 8.2.3 the structure of the URLs and the longevity of the underlying report were considered. Structured URLs facilitate testing, as does consistency in being able to predict and/or determine the contents of the report.

Examples of testability in this research included:

- Deploying the Zipternet micro-experiment to ten user accounts and devices in Google Play Console. The testability was very poor in Google Play Console and the results varied without explanation. There were no sources of guidance on how to test or evaluate Google Play Console or Android Vitals and Google Engineering refused to be drawn on providing any advice or rules on what testing they would permit.
- Testing Iteratively’s service. They supported the testing and were interested in the results and outcomes.
8.5 Cross-cutting topics

Two cross-cutting topics emerged as findings: fidelity of mobile analytics and ethical considerations, each are addressed in turn.

8.5.1 Fidelity

Moonpig accepted that not all crashes that end-users experience will be reported as the reporting is optional.

An interesting phenomenon was observed during the Catrobat hackathon for Pocket Code where some of the crashes that appeared in Android Vitals were believed to come from ‘soft errors’ in the Pocket Code app. The issue, CATROID-426, was logged during the hackathon and the developers wrote two sets of code changes (also known as ‘commits’). These were merged into the app’s codebase on 21st Nov 2019 and released in the Pocket Code app several weeks later.

The intent was laudable, however, at least some of the soft crashes continued to occur over a month later, as documented in jira.catrob.at/browse/CATROID-422. This issue was raised in the hackathon and closed as a duplicate by one of the developers involved in trying to stop the soft errors from appearing in Android Vitals.

The failure rate varies for crashes and ANRs in Google Play Console’s Release Management reports during the rollout of a release. An example
from StackOverflow [306], shown in Figure 8.14 reflects similar behaviours observed in the commercial, C1, case study. In that case study, the latency of the various graphs in the reports was less than one hour, in other words the reports were under one hour old.

This was key as a rollout to 10% of the userbase involved 1000,000’s of end user devices receiving the updated app; if there were major flaws in the new release 100,000’s of users would have it and need to use it. There was no easy or quick way to replace the flawed new version; a new release would need to be developed and made available to the entire userbase as the more was no more granular way to target updates to users of a particular release.

Currently the release management reports raise more questions than they answer, for example, is there any correlation between when users adopt a new release and the failure rate they were experiencing in a previous release?

Google Engineering’s perspective: In email correspondence in January 2020 one of the Engineering Managers at Google for Google Play Console and Android Vitals observed:

“Crashlytics doesn’t report the same data as Vitals, as we’ve already discussed. Each sees things that the other doesn’t: Crashlytics cannot see any start-up crashes nor ANRs, but it is able to count developer-defined issues beyond crashes. Furthermore Vitals is opt-in by Android users, whereas Crashlytics is SDK crashes so consent comes from installing the app. To reconcile the two yourself you would have to have access to detailed user counts including permissions, but on the Vitals side this is user data not developer data (Crashlytics is different) and our privacy policy prevents us being too granular about this information with you.”

This conversation came about partly from the experiences in 2019 of using Fabric Crashlytics in Catrobat’s Pocket Code Android app, and partly from the following mini-experiment.

Travel Europe app micro-experiment: a locally developed Android app called Travel Europe was downloaded on to 10 Android devices each with a unique Google account and with app usage and diagnostics enabled. One of the aims of the mini-experiment was to determine how completely an in-app analytics service, Microsoft App Center, and the platform analytics, Android Vitals, would reflect the usage and any errors. Flaws were found in both mobile analytics services. The experiment is described in more detail in C. Micro-experiments.

In grey data, Various developers have also observed widely different results in counts provided by various mobile analytics services [307, 308]. Mismatches have also been reported, and analysed [309], for an iOS app.
8.5.2 Ethical considerations

The data collected by mobile analytics may have ethical implications a) for the operator/provider of the service, b) for their partners and customers, c) for the developers, d) for end-users. In this research our main focus is on the implications for the developers, nonetheless the other aspects are also important.

8.6 Improvements to mobile analytics tools

From the findings identified in this chapter various improvements to the tools have emerged, these are covered in this section.

8.6.1 Improvements to Google Play Console with Android Vitals

The research identified several potential improvements to the Google Play Console and Android Vitals, the main analytics tools used by all the projects studied for the app-centric cases. These were highlighted during semi-structured interviews with different app-centric project team members and from the analysis performed during this research, including the worked examples in Appendix A in Sections A.4 and A.5.

Direct quotes from the CTO of Moodspace (June 2019): “As for several things I think are missing:”

- “A gradle plugin to integrate play store uploading into CI processes. I currently use a 3rd party plugin to do this, but it would feel a little more secure if it came from Google.”
- “Top line core vitals figures even if you don’t have enough users!”
- “Someway for testers to download old apks from either internal app sharing, or the internal release track.”

And “Crashlytics only covers the crash report of Android vitals, so unfortunately there’s no way to get things like battery usage of ANR reports unless Google makes those reports available. In terms of crashes, I’d always prefer Crashlytics to Android vitals, simply because there are added features like non-fatal reporting and logs which can make surfacing the cause of errors much easier (but do take need added effort to integrate compared to android vitals).”

Self-evidently, were Google to address the flaws including those listed in Table 8.1, particularly to encourage and facilitate testing and evaluation of their code and service that would help increase the trustworthiness and dependability of their service (as discussed in Section 8.4).

8.6.2 Improving integration

There are several aspects of integration to consider. The first is the provision of APIs rather than web-scrapping; the second is persistent and timestamped links to reports (c.f. how github and wikipedia provide versioned links).
In late 2020 Google made various changes to Google Play Console, they provided the ability for developers to directly download individual stack traces for crashes, something requested several years earlier by other developers [310]. This ability is valuable as it makes it easier to directly obtain the stack trace (and both Eclipse and IntelliJ can process the stack trace to show the relevant lines of code in the GUI). None of the mobile analytics tools offer versioned links, whereas GitHub, the Internet Archive, and Wikipedia do. Versioned links would enable persistent references to the report as it appeared at that time. By example, a versioned link used by the Internet Archive is: https://web.archive.org/web/20160920175338/http://www.crashprobe.com/ios/, and for GitHub is https://github.com/bitstadium/crashprobe.github.com/commit/4398b8ee263d222ed4d55e1dc659d67de11bfaaa. The Internet Archive embeds the date and time in the URL while GitHub uses the hash that was generated for a specific commit to the code repository. It should be possible to have something similar for mobile analytics reports, and if so, then these links would provide a persistent reference to that report at that point in time.

After the end of the cases studies, nonetheless of relevance, in June 2022 Google released Canary 3 of the integrated development environment (IDE) called Android Studio Electric Eel. This includes the facility to directly see and work with crashes reported by Firebase Crashlytics within the IDE. This makes the analytics information immediately and continuously available to the developers rather than relying on them visiting the Crashlytics website. It aims to reduce context switching and also encourage faster investigation and remediation of crashes.

Sentry provides various tools to help development teams focus on key issues, described in blog.sentry.io/2021/04/20/silencing-distractions-with-review-list-and-automations. One tool is integration with bug tracking systems including Jira and GitHub. The second assesses the health of new app releases.

Several aspects of improving the integration of mobile analytics into development practices. These include:

1. The ease of sharing pertinent information between the mobile analytics service, issue tracking, and to the developer’s IDE. Similarly many provide integration with various collaboration tools such as Slack, Microsoft Teams, and so on. At Google IO 2022, Google announced integration between Firebase Crashlytics and Android Studio [311], and Firebase Crashlytics with Google Play [Console] - both are intended to help developers to work more efficiently and effectively with crashes reported by Firebase Crashlytics [312].

2. Ways mobile analytics reporting could help developers to tackle and fix more of the failures. The GTAF case study provided an insightful example where the developers sometimes avoided addressing ‘difficult’ failures. Perhaps, a service similar to GitHub’s auto-suggestions integrated into mobile analytics and the developer’s IDE could help reduce risk of exacerbating a situation by taking action (inaction is sometimes perceived as a safer course of behaviour than trying to tackle an issue and being seen to fail to do so).

3. Cohesive tracking and support from design, through implementation, to deployment of the use of mobile analytics. An approach...

[310]: R15 (2018), How can I get app crash log from Google play console
[311]: Android Project (2022), App Quality Insights from Firebase Crashlytics
[312]: Crowe (2022), What’s new from Firebase at Google I/O 2022
8.6.3 Improving the auditability and verifiability

Full end-to-end auditability and verifiability of a mobile analytics toolchain can help increase the trustworthiness of the toolchain from use of any client-side SDK through to the reporting (including any interaction via APIs, being provided for reporting, analysis, configuration, and interaction).

This is necessary because in Section 8.4.1. Flaws in the mobile analytics tools and/or services various flaws were identified in several of the mobile analytics services. And in Section 8.2.5. Which apps can Android Vitals help? some of the additional restrictions were discussed that demonstrated that Android Vitals provides little information of value for apps with user-bases of less than a few thousand. Furthermore, a small experiment, Section 8.5.1. Fidelity, was sufficient to indicate additional concerns in the fidelity of the mobile analytics tools.

Open source testing of mobile analytics SDKs: The company who developed what became part of Microsoft App Center’s crash reporting, Bit Stadium GmbH, developed a set of opensource projects called CrashProbe which they and various other providers of crash reporting SDKs used to test the behaviours of their respective SDK. An example of their report, from 6th September 2016, is in Figure 8.15. The report was last updated on 8th Jun 2017, however the site remained online for another 3 years, until 20th Aug 2020, before disappearing.

8.6.4 Improving the analytics performed

The analytics on offer in the current tools are basic and the predetermined groupings are fixed by the tool / service provider. For the enterprise-level paid-for services Google and Microsoft provides mechanisms for the data to be exported for their in-app analytics (but not for Android Vitals or Google Play Console, as far as I can tell) \(^{27}\). It is not clear to what extent enterprise customers can configure the analytics that’s performed, there was no opportunity to evaluate it in the Industrial case study, C1.\(^ {28}\)
**Future design?** The introduction of more configuration options in mobile analytics tools may be useful in terms of a) grouping (e.g. to define the groupings for ‘similar’ failure clusters) and b) comparisons (e.g. compared to what? what are the commonalities and contrasts between when the app’s measured as performing well vs. performing badly?)

Simple facilities such as the ability to search through the failures to find any failure clusters that match would be useful. A recent example is searching for instances of an `IndexOutOfBoundsException` in the Kiwix custom apps\(^{29}\) where Android Vitals had to be checked page by page for each app to see if the crash was still happening rather than being able to perform a search of the results online.

Aggregation and mining across the matching clusters would also be useful. Similarly, tagging/labelling might also help improve the utility of the reports, *ditto* facilities to cross-reference within and across systems (c.f. hyperlinking and reference links). Another aspect would be helping developers recognise common causes and common failures, and the likely effects of addressing these failures in future.

\(^{29}\) Index Out of Bounds Exception on Custom App #2542
### Results for iOS

The following chart shows the overview of all tests:

<table>
<thead>
<tr>
<th>Provider</th>
<th>ARmv7</th>
<th>ARm64</th>
</tr>
</thead>
<tbody>
<tr>
<td>HockeyApp</td>
<td>IPod touch 5th gen, iOS 9.2</td>
<td>iPhone 6S, iOS 9.2</td>
</tr>
<tr>
<td>Bugsnag</td>
<td>IPod Touch 5th gen, iOS 9.2.1</td>
<td>iPhone 6, iOS 9.2.1</td>
</tr>
<tr>
<td>Apple</td>
<td>iPad 2, iOS 8.3</td>
<td>iPhone 6, iOS 8.3</td>
</tr>
<tr>
<td>Crashlytics</td>
<td>IPod touch 5th gen, iOS 9.2</td>
<td>iPhone 6S, iOS 9.2</td>
</tr>
<tr>
<td>Raygun</td>
<td>iPad Mini, iOS 9.3.5</td>
<td>iPhone 5S, iOS 9.3.5</td>
</tr>
<tr>
<td>Crittercism</td>
<td>IPod touch, iOS 8.1</td>
<td>iPhone 5S, iOS 8.1</td>
</tr>
<tr>
<td>NewRelic</td>
<td>IPod touch, iOS 8.1</td>
<td>iPhone 5S, iOS 8.1</td>
</tr>
<tr>
<td>Splunk MINT Express</td>
<td>IPod touch, iOS 8.1</td>
<td>iPhone 5S, iOS 8.1</td>
</tr>
</tbody>
</table>

**Figure 8.15:** Crashprobe iOS SDK results from 27th Nov 2016. Source: [https://web.archive.org/web/2016127/155426/http://www.crashprobe.com/ios](https://web.archive.org/web/2016127/155426/http://www.crashprobe.com/ios)
8.7 Chapter summary

This chapter has explored the use and potential improvements to mobile analytics tools:

▶ **Design**: This encompasses SDK design and developer experience. An SDK encapsulated failures in React Native to the extent that the platform (Android) did not record them. SDKs also collect meta-data that helps developers find and fix the cause of failures. There are numerous engineering challenges for developers of mobile analytics SDKs. Focus on developer experience is vital for providers of mobile analytics. Iteratively innovated by providing tooling aimed at improving the coherence and consistency of mobile analytics.

▶ **Fitness-for-purpose**: Mobile analytics products need to provide a good product fit for app developers, including providing actionable reports and integration into developer workflows. Programmatic access to analytics outputs is particularly important for large organisations and development teams. Android Vitals, while ubiquitous, only helps developers once their apps generate sufficient data to pass undocumented thresholds within the product.

▶ **Utility**: This includes the efficacy of the mobile analytics tool(s), the benefits of combining tools, and bug-localisation. Mobile analytics needs to provide value to developers in terms of finding and addressing reliability issues.

▶ **Dependability**: This considers the extent to which developers can rely on a mobile-analytics service and/or the underlying tool; and even a tool with few flaws cannot compensate for a heavily-flawed service. There are numerous flaws in the mobile analytics tools and services; 17 were identified in Google Play Console with Android Vitals, the most studied of the offerings. Link rot and testability also adversely affect the dependability of the offerings.

▶ **Cross-cutting issues**: These include fidelity and ethical considerations.

▶ **Improvements**: As Google Play Console with Android Vitals was the most-studied of the mobile analytics services, most of the improvements were aimed at improving it. In addition to addressing the various flaws identified through the research, Moodspace requested various improvements to the service it provides in order to increase the value it provided them. Additional improvements were identified in terms of integration, auditing, verifiability, and in the actual analytics performed.

Despite the various flaws identified in the various mobile analytics tools, the evidence from the case studies highlights many examples of how these tools helped the various development teams address a subset of the issues. Nonetheless, there is plenty of scope to improve the efficacy of the tools, particularly in relation to their integration, trustworthiness, and the range of analysis functions supported. As noted in the previous chapter, bugs in some mobile analytics SDKs meant the experience of using them was worse than not using them.
Discussion, Conclusion, and Future Work
Apps are a popular and relevant subset of all software. They run remotely on other people's equipment where they are the primary owners of the data on their devices, and where the platform and pre-installed platform software determine various aspects of the data collection. The apps and the analytics libraries they use control what data is reported and when.

This research has shown that when developers apply mobile analytics to find failures they are able to significantly improve the stability/reliability of their mobile app. When they stop paying attention and don't apply mobile analytics entropy returns eventually for a variety of reasons including new releases of the operating system, updates to third-party software used by the app, such as the Android System WebView, and flaws in ongoing software development and maintenance of the app. The developers do not need to address all the issues that are reported to effect material improvements (and indeed some issues are impractical for developers to fix in the time and resources they have available).

When developers choose to include in-app mobile analytics they generally prefer to use those mobile analytics as their primary source of information, even if the platform-level analytics also finds a common subset of the same failures. The meta-data collected and reported by the in-app mobile analytics combined with the availability of reports even at low usage volumes of the app are key drivers in this preference to use the in-app mobile analytics reporting. For two of the app-centric case studies the data collected by in-app mobile analytics SDKs was sufficiently intrusive that they chose not to use in-app mobile analytics in their apps. Some projects choose to have several in-app SDKs embedded in their apps, often for distinct purposes.

The final chapter of this thesis presents a focused discussion of the findings with respect to the different facets of the overall research question that has been investigated. This is followed by a summary of the contributions of this research before concluding with some thoughts on future research activities to explore the area of mobile analytics further.

9.1 Discussion

As a recap, the research has been investigating how the application of mobile analytics in software development practice can improve the reliability of mobile apps. Based on this research question, the discussion section that follows covers three groups of topics. The first group considers
the application of mobile analytics in terms of utility and scalability; the second group considers software development practice in terms of empowering developers and empowering users; and the final grouping considers the reliability of mobile analytics tools in terms of measurement reliability and improving the capabilities of the analytics.

The research reported here is exploratory in nature, rather than confirming specific hypotheses with respect to the use and effect of mobile analytics. Therefore, the discussion of the findings is not conclusive but instead looks to provide a better understanding of the use of mobile analytics and the characteristics of analytics tools. Based on this understanding, each section that follows concludes with a discussion of the implications and suggests some hypotheses for further investigation.

9.1.1 Applying mobile analytics

Applying mobile analytics includes two considerations: utility and scalability. In this context, utility refers to both how developers use mobile analytics and the usefulness of analytics data for improving app reliability. Utility is one of the high-level themes in Chapter 8. When reflecting on the findings from the research, scalability also has two aspects, namely the ability to deal with large volumes of data and a variety of device types, as well how the findings scale beyond Google Play to other mobile app ecosystems. Scalability is a theme from Chapter 6 and the discussion builds on the theme being mentioned in Section 6.6 of that chapter. All of these aspects are important for understanding the application of mobile analytics and will be discussed in turn.

Utility

As Chapter 6 indicated, in every one of the app centric case studies developers were able to use the outputs of mobile analytics to successfully find and fix crashes that caused their app(s) to fail in use. This corroborates the work of Patro et al. that showed how two app development teams were able to use mobile analytics over several years to monitor and address multiple issues in their respective mobile apps [146, 147] albeit they used custom in-app analytics rather than platform or commercial mobile analytics services.

The projects in the app-centric case studies ranged from sole-developers to 100+ person teams at a major enterprise. The mobile analytics ranged from purely relying on what the app store/platform provided through projects that included in-app crash and error reporting, to those who incorporated multiple mobile analytics SDKs. Generally, those teams that included at least one mobile analytics SDK found they preferred using that mobile analytics service in preference to the platform provided analytics in so far as the mobile analytics service provided similar information. The possible exception was the GTAF project who weren’t able to provide sufficiently specific answers. There has been numerous research published regarding characteristics and practices of mobile app developers e.g., Francese et al. [313] on app development practices including characteristics of some (non-corporate/industry) mobile app developers, yet none studied the use of mobile analytics. Similarly research into the effects of app stores,
9.1 Discussion

*e.g.*, Martin et al. [314], yet none considered the use of platform-level analytics such as Android Vitals. And research into modern release practices for apps, *e.g.*, by Adams and McIntosh [81] did not consider the release management tools built into Google Play Console nor the utility of in-app mobile analytics even though both were in general use at the time.

Where the in-app analytics service lacked particular types of failure, ANRs is the key example, then the teams also used Android Vitals from time to time. Moonpig demonstrated the benefits of ongoing checking of both the platform-level and in-app analytics, they obtained the benefits of combining several mobile analytics tools (*a theme in Chapter 8*) rather than preferring one over the other. Conversely LocalHalo missed a spike in the failure rate even though they had both in-app and platform level analytics available to the CTO. As highlighted in Chapter 6, they did not notice what was being reported.

This research corroborates arguments presented in [315] where the app store (Google Play) generates actionable reports for developers with the aim of helping developers improve the performance and quality of their apps (Chapter 8). To be fair to Google, their tools in Google Play predate the paper. However the authors might not have had first hand experience of using those tools.

**Implications:** based on these results and from examples drawn from grey materials, using mobile analytics appears to provide utility to many development teams. This research highlights a key development practice, app developers often choose to use in-app mobile analytics including crash and error reporting. (*Choosing mobile analytics is a theme in Chapter 6.*) As such, given the ubiquitous use of in-app and platform-level mobile analytics and their utility for developers one area for further study is to consider what stops app developers from voluntarily actively using mobile analytics on an ongoing basis, why is usage often sporadic and piecemeal? Another area of future research could be to explore ways and times when mobile analytics would provide maximal value to app developers, including the capabilities provided by the tooling (*similar to Iteratively’s design, build, and verification tools*), the design of any SDK (*a theme in Chapter 8*), reporting, alerting, and integration into developers’ working practices, as highlighted in Chapter 8.

**Hypotheses:** The analysis of the findings relating to utility of mobile analytics suggest the following hypotheses for further investigation:

- App developers choose to include mobile analytics yet they seldom use them unless the combined team—including their leadership—see the value in doing so.
- App developers are able to use analytics reports to address some reliability problems with their mobile app, but many flaws go unnoticed.

**Scalability**

The research findings indicate that mobile analytics tools scale to running on billions of end user devices and similarly no practical limits
were encountered in the reporting abilities of the tools and services. In other words, the underlying systems scale. However, the findings of this research highlights that limitations to scalability are often induced by project leads and/or sponsors. Leads who do not provide access to the mobile analytics reporting to their developers, a topic discussed in Chapter 6, Section 6.3.6 and sponsors who choose not to pay for mobile analytics services, for example LocalHalo.

Microsoft Research developed and evaluated highly scalable automated testing for Windows Phone apps using crash data collected from Microsoft’s App Store for these apps [122]. Their work indicated that app stores are uniquely placed to access and use crash data across all the apps in an app store to find the most common flaws in mobile apps, reproduce some of them automatically, and then to help developers reproduce and test for those flaws.

**Beyond Google Play?** This research is limited to apps available in Google Play for practical reasons. Here we consider whether the approach could scale beyond Google Play for Android apps, and later whether the approach could work for other platforms.

Platform providers who collect platform level analytics e.g. Google Android, Huawei Harmony OS, Apple iOS, etc.) have insights into not only the behaviours of the application code, they also have data on the stability of libraries and could use this data to help both library developers and app developers to address flaws related to the libraries being used in apps. Indeed with the launch of the SDK Index, developer.android.com/distribute/sdk-index, Google have recently done so; this extends their work with library developers where they provide crash statistics, etc. [316]. They also provide app developers with various statistics on the top libraries [317].

Pon, et al discussed how three companies: Google, Amazon, and Xiaomi, have developed their own platforms [318].

Additional Android app stores are available, particularly in China as [319] describes. Of the Chinese app stores, in 2018 only 2 (Tencent Myapp, and 360 Market) provided a quality rating. Their work indicates that at least some app stores are not likely to provide stability analytics similar, or equivalent, to those Google provides in Google Play Console and Android Vitals. Developers, therefore, would need to implement any analytics into their app rather than rely on the app store. Two more recent app stores are discussed next: Huawei, and Amazon.

**Huawei App Store:** Key differences between Huawei’s provision and that provided by Google include the lack of integrated platform (device-level) analytics from the Operating System, the need for developers to integrate a full Analytics SDK in order to then use the crash and error reporting SDK. They do not mention the equivalent of Android Vitals. In terms of pre-release checks and testing, they provide various elements that offer similar capabilities to the pre-packaged pre-launch reports that Google Play Console provides. For example, they have an Integration Check [320] which appears to perform various forms of static analysis and Cloud Testing [321] that in turn includes similar automated testing capabilities.
As Huawei develops HarmonyOS and their app store, potentially platform-level analytics could be added and made available by Huawei in future.

**Amazon app store**: Another major app store for Android apps is the Amazon appstore, which includes ‘millions of devices in over 236 countries and territories’. They provide an amazon developer console which offers a subset of the Google Play Console’s features together with some Amazon specific product and service offerings.

Many, but not all, of Google Firebase Android SDKs are able to be used beyond Google Play. The other SDKs require Google Play services which is part of the Google Android platform. And uniquely, firebase-analytics states ‘automatic insights such as demographics are only available on devices with Google Play services.’ [322].

These brief introductions into Huawei’s and Amazon’s, global app store ecosystems indicate they provide sufficient developer-oriented tools and services for developers to be able to measure reliability of their Android-like apps were they distributed in these app stores, and to be able to use the analytics offered by these providers (and potentially other analytics providers, for instance Count.ly claims their Android SDK should work on HarmonyOS [323]) to identify reliability issues and potentially address them. Neither of these ecosystems currently provide the richness of the Google Play Console or Android Vitals, they could choose to do so in future and the underlying Android based operating systems could gather similar data, at least from a technical perspective.

**Implications**: one of the implications of the findings is that many app development teams are self-limiting in terms of scalability in the use of mobile analytics in the app development practices, the exception was Moonpig who provided all the developers with access. With the notable exception of Google Analytics 360 where the recommended pricing starts at $50,000 USD the in-app mobile analytics offerings are either free or low-cost, e.g., Microsoft App Centre’s Analytics is free. A possible area for future study is to understand the motivations of development leads and project sponsors in when and why they choose to provide access and when they chose not to.

**Hypotheses**: the discussion on scalability of mobile analytics suggests the following hypotheses:

- Mobile analytics can be used to improve the reliability of apps in Google Play and in other app stores for Android apps.
- Mobile analytics can [also] be used to improve the reliability of apps for other platforms, including Apple iOS, and Verticals including Healthcare [324, 325].

For both these hypotheses, platform providers choose the extent to which they collect and share platform-level analytics and app developers choose in-app analytics that comply with various policies including those of the platform provider. The platform provider, therefore, has a strong influence in the scope and efficacy of the mobile analytics.
9.1.2 Software development practice

Chapter 6 presented the findings on the processes developers used when they used mobile analytics. The research highlights that developers were able to interpret the mobile analytics reports sufficiently well to address failures in their respective mobile apps, albeit at least some of the developers avoided issues they perceived as difficult e.g. GTAF, as described on 144.

The Catrobat case study in particular provides a conundrum worth exploring further. The team did their best to apply many well regarded software engineering practices even though the team comprised mainly of students who were still learning their craft in terms of software development and software engineering. And yet, they paid little attention to mobile analytics without a product owner, even though they were able to achieve approximately 4x improvement in the crash rate for Pocket Code through applying information reported by Android Vitals.

One possible hypothesis is the Catrobat team were overly focusing their efforts in pre-release activities. Kitchenham, observed that the approach of taking a product view of quality “is frequently adopted by software-metrics advocates, who assume that measuring and controlling internal product properties (internal quality indicators) will result in improved external product behavior (quality in use).” [326, p. 15].

Would working backwards from the failures help the team address the quality in use more effectively?

The research findings have established two clear characteristics of development practices:

1. When developers pay attention to flaws reported in analytics tools they are able to effect improvements to the app which significantly reduce the failure rate and improve the stability of the apps for end users.
2. Developers will release updates that unintentionally and or exacerbate the failure rate, despite their best intentions. They need to pay ongoing attention to the reported failures if they wish to maintain or decrease the measured stability of their apps as both their app and the ecosystems evolve.

Empowering developers

A necessary prerequisite to empowering developers is providing each of them access to the mobile analytics reports. As we discovered, this is seldom the case, as Chapter 6, Section 6.3.6 found. The mobile analytics tools studied in this research offer a read-only option to the reports and access is either free or inexpensive per developer so the in many cases should be practical and a manageable risk for organisations. One hypothesis is that developers would love to use the tools yet they are not encouraged or rewarded to do so in practice (i.e. beyond organisations playing ‘lip-service’ to quality being a mantra. This might be an area worth further study, on the team dynamics, motivations, incentives and disincentives for developers to actively use mobile analytics.

To adopt new practices developers need to be motivated to do so; the motivations may be a mix of intrinsic and/or extrinsic. McAvoy and
Butler described insights they gleaned during a nine-month embedded case study with a development team adopting Agile development practices[327]. They identified several “failures to learn” in the developers. One of the failures was connected to developers being unwilling to risk their work being criticised by their peers (Ibid. pp 38,40. Perhaps similar fears contribute to the piecemeal engagement of some app developers with mobile analytics where the reports effectively identify failures in the developer’s work. Another factor in terms of empowering developers is psychological empowerment[328, pp. 885-887] where the app developers need to believe they are free to take advantage of mobile analytics and spend time checking the ongoing behaviours of their apps in production. Moonpig case study indicated it is viable for a development team to systematically adopt using mobile analytics to maintain high reliability of their app (and service). They confirmed the engineering team had the trust of their management which empowered them to use mobile analytics and other software craftsmanship practices.

One of the tests of whether developers are truly empowered is when they are allowed to delay bug fixes based on their interpretation of the analytics reports to determine the impact to reliability of the app. In Chapter 6 in Section 6.2.1, the Moonpig case study provided an illustrative example of where the developers were able to chose to delay a bug fix until it met other release criteria, such as the release timetable, as discussed in Chapter 3 in Section 3.3.1.

Hackathons [329] proved to be productive ways to foster fast and immediate progress with the application of platform-level mobile analytics, not least because the results were already available to use. Further study in to ways to engender longer term engagement and adoption of mobile analytics in the software development practice may help both research community (in a similar vein to the work of Adams and McIntosh in release engineering [81]) and industry.

**Implications:** developers first need access to mobile analytics, which may require their organisations to fund the cost of direct access to analytics services for multiple members of the team, or establish processes for consistently sharing analytics reports internally. However, access is necessary but not sufficient - developers must also have the motivation and freedom to actually adopt the tooling to enable them to improve and maintain the reliability of their mobile apps.

Interventions like hackathons can help foster this motivation and create spaces for developers to engage their creativity when working with analytics reports. Nolte et al. identified three factors that led to longer term continuation post-hackathon [330]. These are 1) skills matching and 2) diversity of skills when combined with 3) longer term intentions to continue (Ibid p. 145:21). If hackathons to adopt mobile analytics are congruent with their research then hackathons that actively contain these three factors should have longer-term and long-lived adoption of the app developers proactively adopting and using mobile analytics to improve reliability.

**Hypotheses:** the findings highlight hackathons as an important mechanism for empowering developers and suggest the following hypotheses for further exploration:
Hackathons enable app development teams to make fast and efficacious improvements to the reliability of their apps provided the app already has at least one source of crash analytics available to the development team.

Hackathons have a short half-life in their efficacy and the practices of proactively using mobile analytics need to be sustained using additional approaches.

Empowering users

Now may be the ‘golden age’ in terms of app developers being able to default to gathering mobile analytics data, as the legal context may constrain this practice in the future. In particular, legislation may require developers to change the default to not collecting mobile analytics data unless users explicitly opt-in. There may be further constraints on what data can be collected and who is responsible for protecting and preserving the data [331, 332].

Conversely, currently users have little autonomy or control about data being collected and analysed related to their use of mobile apps. Two examples from the app-centric case studies are Pocket Code which presented 727 words, 4,590 characters, when users first started the app, and Moonpig’s more informal approach (see , in Chapter 7, starting on Page 169 for details). And at a platform-level, for example, when users create a new Google account the default one-step activation enables extensive tracking including web and app tracking [331, 332]. A Google account is essential for using an Android device with Google Play so this means that Google, at a platform-level, is likely to collect analytics from the vast majority of Android users about their app usage.

As discussed earlier, the majority of Android apps include at least one mobile analytics library. These apps could offer privacy by default however developers and/or their organisations mainly choose to collect data by default. Some apps offer users an option to control the data collection by mobile analytics SDK(s); in turn the mobile analytics SDK needs to provide developers with a mechanism to disable/enable collection of the underlying data.

As the vast majority of Android apps in Google Play are free the app developers may consider the provision of analytics data as a quid pro quo. And yet there may be plenty of scope to improve the situation for these two key stakeholders in the mobile app ecosystem, who in turn are part of a larger platform. An interesting business-focused discussion on platforms in [35] also applies to various Android platforms (from Google, Amazon, and others) and Apple’s iOS platform.

[331]: BBC News (2022), Google sign-up a 'fast track to surveillance', consumer groups say
[332]: Pant (2022), European consumer groups take action against Google for pushing users towards its surveillance system

[35]: Kapoor et al. (2021), ‘A socio-technical view of platform ecosystems: Systematic review and research agenda’
Figure 9.1: Kiwix: asks user what data they would like to send

and/or the SDK developers could modify the SDK to provide such access. Furthermore, although current mobile analytics SDKs transmit the data transparently and automatically, it is possible to involve the users in the transmission either upon request by the user or more generally. As a simple example the author and the Kiwix project implemented an in-app feedback mechanism where end users can generate a diagnostics report as an email generated by the app. The Kiwix UI to send a diagnostics report is shown in Figure 9.1, it was based on another opensource project Ereza/CustomActivityOnCrash.

The Android platform does not currently provide any documented way, such as an API, for an app to check the user’s setting for providing diagnostics information. If it did, app developers could then make their apps adapt accordingly. Note: The adaption might be more nuanced than simply applying the same setting within the app as users may wish to control the provision of analytics on a per-app or contextual basis, for instance. The lack of such an API means each app developer is responsible for deciding whether to ask user’s for permission or simply assume their app can collect and send analytics data.

Listing 7 is an example Google provides for Android developers to learn how to use the AndroidX preference library. This example generates a GUI to ask users if they wish to enable message notifications and/or send feedback including reporting technical issues. Given this documented example there are no technical reasons why similar dialogues couldn’t be added to the Android platform and to individual apps.
Preferences, permissions, and usage analytics share similarities in terms of considerations such as informed consent, whether the settings are temporary or permanent, and so on. Informed consent was discussed earlier, in Chapter 7, starting on Page 169.

Both ‘informed’ and ‘consent’ are important considerations - how to improve (and perhaps even check for) end users being sufficiently informed, and then also providing mechanisms where end users can freely consent to whatever data is gathered for analytical purposes. Or is it enough for developers to deem use of an app as providing sufficient consent? This appears to be an area well worth further study.

Note: other changes to the relationships between end-users, their organisations where appropriate, app-developers, mobile analytics providers, the app store, and platform providers may need to be re-envisaged as changes to the availability of analytics data may upset current business models.

In summary, currently end-users of mobile apps have unnecessarily limited choices in terms of the data collected by app developers or the Android platform. It is practical to improve the service provided to end-users and to provide them with greater freedom and control over the data collection. Some may wish to have access to the underlying data too. The platform could provide app developers with information about the user’s data collection preferences and could also implement changes to help give users more choice while also protecting them from intrusive data gathering (both Google and Apple have made changes in relationship to other tracking mechanisms [333]).

Implications: Users of Google powered Android devices need to know they can actually control whether platform-level analytics is harvested from their device(s).

App stores, including Google Play, could mandate that apps that incorporate analytics should provide first-class functionality to enable end users to choose whether they wish to opt-in to mobile analytics. Google Android provides a similar mechanism for in-app reviews 8. It could also provide developers with an API that provides the value of the setting(s) which would enable app developers to respect the choices of end users.

[333]: Wakefield (2022), Google moves to make Android apps more private

8: Google Play In-App Reviews API
Hypotheses: the findings highlight that our understanding of the needs of the user when integrating mobile analytics could be developed by exploring the following hypotheses:

- Few users of Android know of their options to control whether Android collects usage and performance analytics. Therefore, the majority contribute by default.
- It is possible for even well-meaning apps to discourage end users from using their apps owing to design and communications challenges in the provision of in-app informed consent.

9.1.3 Reliability of mobile analytics tools

As discussed in the previous sections, this research identified ways in which developers use mobile analytics to improve the reliability of their apps. Additionally, it highlighted some of the difficulties in using analytics for this purpose, due to limitations in the reliability of the mobile analytics tools themselves.

Measurement reliability

Various findings lead to questioning the reliability of how mobile analytics measures and reports the data that their SDKs collect from mobile devices. In Chapter 8, Section 8.4.1, seventeen distinct flaws were identified in Google Play Console and Android Vitals together with additional flaws found in Microsoft App Center and Azetone resulting in at least nineteen distinct flaws in addition to any documented limitations e.g. in the respective SDKs. Further inconsistencies were discovered when deploying one of the micro experiment to ten user accounts and devices where some devices but not all appeared in the reports.

At one extreme the reliability of the mobile analytics offerings may be sufficient to meet the needs of virtually all of the mobile app developers as none of those from the case studies questioned or were concerned about any flaws they encountered and they were able to fix issues and see improved results. At the other extreme, given Google in particular holds developers to account for the reliability of their apps, the underlying systems should be demonstrably and openly assessed for their reliability, availability, performance, and so on.

With the availability of various opensource mobile analytics offerings, including Count.ly, PostHog, Segment, and Sentry, they provide an opportunity for further study; and for that study to openly evaluate and publish the results of the tests and evaluation results (subject to any licensing restrictions, etc.).

Implications: The reliability of the measurement instruments (the SDKs, the data transmission and collection mechanisms, and the reporting) are still relatively under-researched entities. Google appear unwilling to subject their work to the standards they apply to the developers in their ecosystem. Given the dominance of major platform providers flaws and miscalculations can have material and adverse effects. The rapid developments in AI have led to a growing call for algorithmic decision
processes to be held to account. This also need also applies to mobile app platforms including their data collection and analytics [334]. They should demonstrate they are worthy of trust in their systems and their practices, offer the right to redress, and be held to account [335].

**Hypotheses:** the findings relating to the reliability of mobile analytics measurements suggest the following hypotheses for further investigation:

- Publishing the source code and the algorithms, will reduce the scale and significance of measurement flaws.
- It is possible to use open source approaches to mobile analytics tool development without adversely affecting the commercial viability of these products.

**Improving analytics capabilities**

As discussed in Chapter 7 in Section 7.3.2, during the period of the research Android was enhanced so the platform could provide failure data it had detected to apps that requested the data for their app (apps could not obtain failure data for other apps on the same device). This was adopted by various in-app mobile SDKs to augment their analytics capabilities.

A trend first observed during this research with HP’s AppPulse Mobile was the automated instrumentation of mobile apps to collect app lifecycle and meta-data without the app developer needing to write any code to explicitly collect these data. Various mobile SDKs, including Sentry, now include automated instrumentation of these and other data. Other data includes Network requests and responses, a topic covered earlier starting on Page 182. Segment presented the results of their newer SDKs where they had improved the performance and dependability several fold.

The flexibility and UX of the reporting aspects of the mobile analytics tools were constrained by what their providers were willing to offer. None provided easy, ad-hoc, query, or search capabilities. Mobile analytics that targeted enterprise customers provided direct data export; and Microsoft App Center’s APIs provided access to much of the analytics data (starting on page 196 in Chapter 8). Data exports and API access both facilitate the extension of mobile analytics into developer (and business) workflows and may increase the utility and value of the analytics data.

A final summary point is the analytics needs to have low latency in order to be actionable. The example in Chapter 8 in Section 8.5.1 indicates a latency of under one hour was necessary, lower latency might provide additional value for app developers. Work by Padmanabha et al. in nearby domains argues that latency can have an adverse financial impact and slow down troubleshooting [336].

**Implications:** The data collection capabilities continue to be enhanced and good ideas are adopted by other providers of other mobile analytics SDKs. Constraints in the reporting systems may limit the value of applying mobile analytics, as may latency in the end-to-end processing of analytics data.
This research did not go into depth in terms of tradeoffs between latency and the value of the analytics, this might be a topic for future research. There is research in another field, in research decisions in the field of human health economics, which considers the expected value of information (EVI), this research might also be of use when considering the value of information provided by mobile analytics.

Hypotheses: the findings inform our understanding of the limitations in the capabilities of mobile analytics tools and suggest the following hypotheses for further exploration:

- Data collected by auto-instrumentation in the mobile analytics SDKs is useful for app developers and can help them troubleshoot issues effectively.
- Latency of the end-to-end service is inversely correlated to value that can be obtained from the analytics.

9.2 Summary of contributions

In this section we revisit the research questions, first with the subsidiary questions, from Section 1.3. Research Questions, to provide an overview of how the findings and discussion address each of the six perspectives. The questions are reproduced here with a summary of the contributions to knowledge for each of the six perspectives, identified here using numerals: 1 - processes, 2 - apps and their artefacts, and 3 - tools and their artefacts; and letters: A for current practices and B for improvements:

1. What do app developers say they do? Findings: App developers have found mobile analytics useful. They vary in how they use them, some do so in extremis so they can avert excessive failures, more generally they are used irregularly as and when they have the opportunity or need to do so. Moonpig used them on an ongoing, strategic basis and were rewarded by maintaining high reliability of their app with a correlation of business success as the business revenues grew and the company went Public (IPO'd).

The human motivations that determine the use of mobile analytics are at least as interesting as the quantitative results of the effects of whatever use occurs. With the exception of the Moonpig development team, none of the teams used mobile analytics on an ongoing, proactive basis.

2. What’s possible in terms of improving their processes, their practices? Findings: As Moonpig and Catrobat project demonstrated it is viable to integrate mobile analytics into development practices. In the case of Moonpig, they added additional logging to the app (using Firebase Analytics) when they noticed or discovered new issues and/or areas of concern; they had developers on-point on-rotation to monitor and make initial decisions when fresh issues were observed in the mobile analytics, and they actively managed their fixes and releases using a composite of business and engineering risk assessments vs. benefits. While the Catrobat app-centric case study also highlighted benefits of analytics for improving processes and practices, it also identified the importance of
projects having key personnel (e.g., product owners) who are responsible for maintaining these practices.

App developers are likely to materially improve the quality of their mobile apps when they actively engage with mobile analytics (whether in-app or platform generated), and even those who use in-app analytics extensively would benefit from actively checking the platform generated analytics as these capture issues and failures that in-app analytics do not. For example, GTAF were motivated to use mobile analytics strategically after their involvement in the case study where they focused on improving the quality of their various apps.

Designing the apps so they emit pertinent information via mobile analytics (such as the approach used by Moonpig) may increase the value of the information being reported by the mobile analytics. One of the mini-experiments corroborates a similar approach to augment information reported by Crashlytics; details are in the appendices in Section C.1. Adding meta-data to crash logs.

1b. What does their source code (and other available development artefacts) tell us about their use of mobile analytics? **Findings:** Over 46% of the active Android app codebases studied on GitHub simply initialised Firebase Analytics and did not use any other feature of the SDK (These projects were a tiny percentage of the open-source Android apps found on GitHub, the majority did not incorporate mobile analytics). All the commercial apps used mobile analytics extensively, while the Catrobat and Kiwix projects chose emphatically not to use in-app mobile analytics. These results indicate commercial app developers rely on in-app mobile analytics for various reasons including for detecting failures of their apps while the app is in use. Services such as Google’s Android Vitals provide, literally, a vital service for app developers, especially those who do not use in-app analytics.

All of the app-centric case studies recorded failures reported by mobile analytics in their issue-tracking system; none did so universally.

2b. What’s possible in terms of improving the product (and particularly the mobile app) through the application/use of mobile analytics? **Findings:** Tools such as those provided by Amplitude and originally developed by Iteratively can help developers improve the design and implementation of their use of mobile analytics as they provide a clear, coherent, end-to-end view of how mobile analytics have been implemented, the custom data the analytics is designed to collect, and they confirm the implementation includes all the custom events that have been designed.

The use of logging, exemplified by Moonpig, to help diagnose issues in the field can help app developers to learn about the factors and conditions that lead to the app failing in use. Mobile analytics is able to measure improvements in the reliability and stability of mobile apps, so developers and their teams have objective, quantitative measurements (allowing for factors that affect the actual failure rate).

The Kiwix project was able to achieve ten-fold improvements in the reliability and stability of their suite of apps. Catrobat was able to effect quick and easy improvements for several releases, and the commercial,
C1 project was able to quickly and effectively tame rogue releases using the release management reports. They also were able to create automated tests that reproduced a complex situation to prove the proposed fix was going to address the failures that caused the app to crash frequently.

1c. What do we learn about various current mobile analytics tools? Findings: There appears to be a general, perhaps misplaced, trust that the mobile analytics tools are accurate and complete - both by the app developers and in the literature. Nineteen distinct flaws were found in the mobile analytics services, as discussed in Chapter 8 in Section 8.4. There are probably many more flaws waiting to be discovered given the practical limitations placed by both industrial collaboration and the scope of a PhD.

Nonetheless the current tools are useful, many are undergoing active and ongoing development, so there is room for optimism in the quality and utility of these tools.

2c. What improvements are possible for mobile analytics tools based on what was learned in the various case studies? Findings: Tools such as the ones derived from Iteratively’s work are subject to ongoing development. They do not currently include failure data so there is plenty of scope for improvements in these types of tools. The open nature of Amplitude, PostHog, Sentry, and other providers of mobile analytics tools indicates they actively seek and accept contributions to help improve their tools and supporting materials and services.

If the tools providers willing to accept the challenge of making their design and implementation testable, auditable, and so on, then they would provide an opportunity to increase the trust and trustworthiness in their tools and services. Furthermore the results and any competition might encourage them to further improve their tools and services.

Finally, let us consider the the core question the research aimed to consider, which is restated here for ease of reference:

How can applying mobile analytics in software development practice improve the reliability of mobile apps? 10.

Drawing on the summary of how each of the sub-questions have been addressed through this research, we can conclude that that overarching question as also answered. Specifically, this research has provided real-world, concrete results of how the application of mobile analytics has improved software development practices, the apps and related artefacts, and of the state of a variety of mobile analytics tools and services. The research has contributed to the understanding of tools and information seldom available to research – of professional app developers, their artefacts, and of professional mobile analytics tools and services. It has also contributed examples of how mobile analytics is being used to help and augment software testing and the quality of mobile apps.

The field of mobile analytics evolved throughout the research period and it is likely to continue to evolve for years to come. The reporting and analysis in the various mobile analytics tools appears to be fairly perfunctory and there may be significant scope to improve the analysis and reporting.
9.3 Future Work

We have systematically explored the phenomena relating to the effect of mobile analytics on development processes and artefacts, and here we use these findings to discuss areas for further study. Four areas of future work are considered. The first three draw on hypotheses identified in the Discussion Section, and the final area builds on one of the themes in Chapter 6 to consider and combine alternative sources of failures, by combining human- and automated mobile analytics feedback.

9.3.1 Investigation of additional app ecosystems

One of the hypotheses in the Discussion section is that mobile analytics can be used to improve the reliability of apps in other ecosystems, including Apple’s iOS platform. There are likely to be many distinctions as each ecosystem is distinct and unique. This leads to four related questions in terms of future work in this area; i.e., 1) what are the engineering challenges and realities in those app ecosystems and what sort of analytics do those platforms provide? 2) What do the apps use? 3) To what extent can the app developers rely on analytics to learn of instabilities or failures in their apps? And, 4) can they apply similar techniques to those described in this research?

9.3.2 Establishing transparency and trust in mobile analytics

Building on some of the background material, in Chapter 2 in Section 2.1.1, and the discussions on empowering users and on measurement reliability a key area for future work is to consider transparency and trust in mobile analytics services and the underlying software.

There are existing fragments that provide elements of transparency and trust in mobile analytics tools and services, for example, opensource SDKs and servers, the CrashScope initiative, Google and many others who are involved in the OpenMined initiative [338] where there are tools and the willingness to establish trust in the design and implementation of data-based research and analysis. This is an area where the combination of industry support and academic research may bear much fruit.

The following list of considerations have been highlighted during the research as being relevant to addressing concerns of trust and transparency when using mobile analytics. Further investigation is required to get a better understanding of how they impact mobile app development practice and the quality of the apps produced.

- **Privacy**: protecting the privacy of the end users.
- **Informed Consent**: encouraging and helping users to make informed decisions about any data collection for analytics. Also, where and how can the User Experience (UX) be improved?
- **Ownership and uses**: especially in terms of who owns the data and who gets to use it?
- **Stewardship**: Impact(s) of having access to sensitive and valuable data.
9.3 Future Work

9.3.3 Operational considerations when using Mobile Analytics

In addition to research into establishing transparency and trust there are various operational considerations that warrant further research. These build on the topics of empowering developers and on improving the analytics capabilities. Topics to research include:

- **Trust**: [Over] trust in decisions made by technology.

- **Sufficiency**: in the context of collecting sufficient to enable us to achieve our objectives of improving our software and our processes.

- **Costs**: financial, data, privacy, performance, bloat. These are closely aligned to performance aspects.

- **Performance**: which includes runtime overhead, transmission overhead, latency and scalability.

9.3.4 Combining human- and mobile- analytics feedback

Android app developers have good access to both platform analytics, via Android Vitals, and various online Review Analysis tools, illustrated in Figures 9.2 and 9.3. Both sources of information flow via the app store, as illustrated in Chapter 2 in Section 2.1.2. The first, platform-level analytics is implicit feedback, the second - reviews - is explicit feedback; they are both feedback mechanisms.

To the best of my knowledge there has not been material research either into the Review Analysis tools, nor into any symbiosis between the reviews, review analysis, or mobile analytics. Such research may provide insights into the review analysis (which may have its own flaws and characteristics) and the extent to which mobile analytics and reviews provide actionable information to developers.

**Symbiotic information sources**: Development teams can use both mobile analytics and user feedback where they complement each other. There was only one example, from Moonpig in Chapter 6 in Section 6.6 where the two sources were combined and contrasted, there may have been other instances but they were not captured in the research.

An useful example from grey literature of how the two sources of information have helped developers is described in [339]. The meta-information Google Android collected when the user wrote the review indicated the user’s device had a low screen resolution. The developer was able to recreate an Android virtual device with the same screen dimensions and reproduce the bug. They then added a specific screen layout suitable for this and other low screen resolution devices.

Existing research by Microsoft in software analytics found that Data/Metrics and Customer input were both deemed important by both managers and developers which indicate the potential of combining both these sources of information. [30, p. 989]
Reviews analysis

View common themes across your reviews, and see how your feedback

**Updated ratings**

<table>
<thead>
<tr>
<th>With replies</th>
<th>Without replies</th>
</tr>
</thead>
<tbody>
<tr>
<td>+2.333 ★</td>
<td>-0.212 ★</td>
</tr>
</tbody>
</table>

**Figure 9.2**: Example of Review Analysis in Google Play Console for the Physics and Chemistry simulations app

Symbiotic information sources would be a fruitful area of future work, and is discussed in Section 9.3.4. Combining human- and mobile- analytics feedback.

The overall development process for mobile apps includes both the release and operational stages of a software lifecycle. Recent research aims to determine whether releases are good, bad, or neutral. One of their key observations is “Android developers need to pay attention to the quality of their app next release” [340, p. 31]. They plan to extend their APP-TRACKER software tool so it can be integrated into the development pipeline for Android apps which sounds promising, especially if the underlying software could also use mobile analytics data as part of the scoring of a potential release.

[340]: Saidani et al. (2022), ‘Tracking Bad Updates in Mobile Apps: A Search-based Approach’
9.4 Final thoughts

Before this research was undertaken, we didn’t understand how mobile analytics might help developers improve the quality of their mobile apps. Through a combination of real-world case-studies, by working with the providers of various mobile analytics tools, and through additional experiments and research, we gathered a variety of data to help address this gap in knowledge. As a result, we now have a better understanding of how mobile analytics are used by app developers in their practices and processes, of the effects on the apps and their artefacts, and have discovered a great deal about the tools, including their utility despite various flaws in the tools and services. In all three areas we have also identified improvements relevant to each area.

The research also provides evidence about the utility and effectiveness about using mobile analytics to improve the quality of mobile apps in a real world context. It is hoped this encourages and inspires greater use of mobile analytics by app developers to improve the quality of their mobile apps and further research into this important area.

The future work identified in this section incorporates recommendations for areas to be investigated by researchers and practitioners, including app developers and tool developers. I highlight both groups in this context based on my experience as a practitioner as well as a researcher. My current role as the CTO of a startup using computer vision encompasses several of these areas of future work, including the use of in-app analytics, Operational Analytics, the design and implementation of our SDKs and analytics service, etc. This opportunity to continue with practitioner based research will enable additional results to be published as the work matures. I hope the overall approach and experience reported here will inspire other practitioners to further develop our understanding of the use and effect of mobile analytics for improving mobile applications.
Context

This appendix contains a range of worked examples that are provided for the interested reader; they illustrate the different forms of data collection and analysis that led to the findings reported in this dissertation. The mappings of findings are maintained using the spreadsheet design described in B. Thematic Analysis.

Structure of each example

The examples follow a consistent workflow, which has been simplified to make the narrative clearer; the actual workflows often included additional iterations, for example as more data was collected (some data arrived on an ongoing basis and bolstered with research-in-progress).

The structure is:

1. Data, including how it was collected,
2. Analysis, including how the data was analysed,
3. Findings, and how they were validated,

A.1 Kiwix experiment incorporating the hackathon in Stockholm

Data collection: The data was collected during the Kiwix Case Study, in Chapter 5, Section 5.6. Screenshots and the failure clusters were initially collected and saved interactively (‘by hand’). Doing so was cumbersome, because of: 1) the ad hoc nature of manual data collection, 2) the effort of navigating through records and extracting content, and 3) the time needed to do so. Hence Vitals Scraper was developed and then used instead to automatically and systematically collect the reports and crash clusters subsequently. It improved the data collection process by at least an order of magnitude in terms of effort and collected all the Crash clusters and ANR clusters consistently.

- Android Vitals on-screen reports showing the failure (crash) rates. These were initially collected manually, and subsequently by using Vitals Scraper.
- Failure clusters were collected using Vitals Scraper.
Listing 8: 1 of 2 crash fixes in Kiwix Android v2.5.3 Source: KiwixMobileActivity.java Lines 1263-1269

```java
if (!mWebViews.isEmpty() &&
currentWebViewIndex < mWebViews.size() &&
mWebViews.get(currentWebViewIndex).getUrl() != null &&
mWebViews.get(currentWebViewIndex).getUrl().equals("file:///android_asset/help.html") &&
mWebViews.get(currentWebViewIndex).findViewById(R.id.get_content_card) != null) {
    mWebViews.get(currentWebViewIndex).findViewById(R.id.get_content_card).setEnabled(true);
}
```

- Release rollout and deployment were tracked using Google Play Console and the integral release management reports that Google Play Console provides during the release of an app and for seven days afterwards.
- Source code pre- and post- changes to address crash clusters were obtained from, and are still available on, the project’s github.com code repository: github.com/kiwix/kiwix-android. GitHub’s counts of lines of code were manually inspected to determine the lines of code that were added/changed/deleted to fix causes of crashes.
- GitHub issue(s) and Pull Requests these were read online on the GitHub website for the entire project. Of those those that contained material related to crashes, ANRs, and/or which included content from Android Vitals were selected for this research: for example Pull Request-1388 which addressed 2 of the most frequent crashes in release 2.5.3
- Contemporaneous notes of conversations with app developers and the project’s technical lead on the likelihood of being able to quickly and safely address the causes of two of the major sources of Crash clusters were recorded as interview notes.

The source code is freely available on GitHub, as are the issues. The developers also maintain a summary of key changes in the app within the same repository [341]. Listing 8 includes one of the two fixes for crashes (it and the other fix are included in Pull Request-1388).

The history of the release of the updated app was preserved in Google Play Console.

Release 2.5.3 was rolled out in Google Play to end users in stages from 19th to 23rd August 2019 in stages, through an overabundance of caution given the small footprint and nature of the changes.

**Analysis**: Analysis included investigation of failures, discussion with key stakeholders, drilling down into identified crash clusters, and feedback on findings from project stakeholders – as discussed in the paragraphs that follow.

Straightforward investigation of the most prevalent failures: this included determining their probable contribution to the crash rate, and identifying the associated source code (identified from the stack traces associated with the crashes). Armed with these details, the researcher sought possible input conditions that might trigger the failure.

This investigation was discussed with the technical lead of the Kiwix project and with three of the app developers. There was consensus throughout that the fixes were self-contained (with a low risk of adversely affecting any other aspect of the app’s behaviour) and that the fixes were likely to be successful. Therefore it was agreed to perform the fixes and release the updated app that week.
Drill-down into details of each of the crash clusters identified the top three most-frequent crashes in the user base across the range of releases being used on Google Android devices that provided app and usage information to Google Play Console. Individual crash clusters, including the stack traces, were reviewed to make sense of the exceptions that caused the app to crash. Across-case comparisons determined that these crashes also occurred in some of the Kiwix custom apps.

Feedback was obtained from the app developers and the project leads who had reviewed the initial analysis and findings. The reports and the design of Android Vitals were also discussed with the engineering team for Google Play Console and Android Vitals. The source code in question (i.e., that matched the stack traces) was reviewed with the app developers.

Findings, and how they were validated: The developers fixed several of the causes of the most frequent crashes with a surprisingly small amount of code of under 25 lines (including 10 lines of text added to the release log). Of interest, at least one of the fixes had actually been made and committed to a pending major release of the app, but wasn’t applied to the current production release until the effects of the crash were made visible using analytics. Details of the crash report and fix are available online Issue 1261. The crash rate stabilised at around 1.1% once the majority of userbase had release 2.5.3 installed; it would have been lower were it not for the UnsatisfiedLinkError exceptions, discussed next.

Several new crash clusters emerged for UnsatisfiedLinkError exceptions. These were not related directly to the crash fixes in the 2.5.x releases; instead they were related to the project choosing to apply advice from Google to use App Bundles [342].

Similar challenges and behaviours exist in code commits; [343] discusses that topic well. Nonetheless, approaches used to detangle commits are unlikely to work as-is for releases, as releases need to satisfy other constraints. Prior research in release planning and release management is discussed in Section 3.3.1.

With App Bundles, Google Play took responsibility for delivering the correct app binaries to suit the end-user’s device’s hardware architecture, e.g., ARM 32-bit, ARM 64-bit, Intel 64-bit, and so on. However, sometimes the users seemed to receive one or more binaries that didn’t run on their device, which led to this crash. What wasn’t well published is that enabling App Bundles is similar to Caesar crossing the Rubicon: there is no turning back. Therefore, rather than having the option to revert to ‘fat binaries’, the project had to find an approach that worked in the context of App Bundles. As the Kiwix Android apps include a native library, written in C++, the solution needed to work for native code in addition to managed code.

There’s quite a detailed issue report available on GitHub.com, Issue 1259 - Crash Report: UnsatisfiedLinkError reported in Android Vitals for 2.5.x users. The cause required in-depth investigation, changes to the build process, and changes to the application code in order to reduce the likelihood of the incorrect binaries being deployed to the end user devices.
Various developers on the Kiwix project continued to make corrective changes to the codebase which made ongoing incremental improvements to the app released to the 2.5.x releases.

The results of the investigation motivated the development team to choose to attempt to address several of flaws that had been discovered through the investigation of the information provided by mobile analytics: Android Vitals. They wanted the main app to be more reliable and their hypothesis was that this might lead to more, satisfied users. The development team was not able to reproduce any of the bugs (the crashes) nonetheless they were confident they would be able to fix the bugs that allowed the crashes to occur.

The team chose not to write any automated tests to exercise the relevant code; their code coverage was low for the app anyway, and they were not in the habit of writing automated tests. They preferred to modify the source code directly for the app and then make a new release of the app to determine whether they had actually fixed the issue.

As the new release was adopted, the crash rate did indeed improve; the developers were justified in their assessment that they would be able to fix the bugs even without being able to reproduce the bugs.

The time needed to perform the investigation, obtain consensus to attempt to fix several of the most frequent crashes, and then to complete the necessary changes to the source code amounted to several days of work.

Creating the new release and managing the rollout was not measured directly. Prior experience with this app indicated that the effort was less than one day, spread over several calendar days. The changes in the source code to fix two crashes were packaged as separate ‘commits’ and consisted of 10 changed or new lines and 12 lines of code being deleted. (Deletions of blank lines did not affect the behaviour of the code and were therefore discounted; and the delete+add to update the value of versionPatch similarly did not affect the behaviour of the app and so was also discounted).

Ongoing reduction in crash rate was measured by reading and drilling-down into Android Vitals reports and crash clusters. These were validated by comparing crash rates for WikiMed and Kiwix over multiple releases of both apps, and Table 7.3 in Chapter 7 provides a comparison of the respective crash rates for the experiment and control apps. The hackathon resulted in the transition from release 1 to release 2 of the Kiwix Android app in Table 7.3.

When the code to fix a crash was applied, the incidence of that crash cluster in the Android Vitals reports decreased in general and in particular for the new release that contained the fixes.

**Conclusions:** There was no material improvement in the crash rates for WikiMed (or other custom Kiwix apps) that did not incorporate the fixes. Hence, the most likely cause for the reduction of a particular crash cluster (or group of crash clusters with a common stack trace) is because a) the fix was applied, and b) it addressed the particular failure sufficiently – i.e., we are applying the principles of concomitant variation [344, pp. 263-264].
A.2 Crashes in cross-platform containers

This example has been chosen because it demonstrates how the use of two mobile analytics tools helped to provide insights into an incident that inverted which of these tools provided the crash reporting.

Note: some of this example is presented in Chapter 8 in Section 2. Runtime encapsulation of failures. Here - for consistency - the research methods, analysis, findings, and conclusions are described.

Data collection: The mobile analytics data was collected from three primary sources: 1) Sentry’s online dashboard, 2) Android Vitals, 3) Sentry’s automated emails. These were augmented by using the Ask the app devs method and by searching various grey data including GitHub for the React Native Expo framework. In particular, issue 5839 [269] appeared pertinent.

Grey literature was also searched as part of investigating the behaviours observed by Sentry and React Native. Android Vitals was compared across LocalHalo and for GTAF’s Taskinator app 5.

A high crash rate was reported in Android Vitals for the production release of the LocalHalo Android app. The top crash cluster seemed to be related to java.lang.RuntimeExceptionhost.exp.exponent.experience.a$b.run. The crash cluster was found by drilling-down into the reports provided in Android Vitals.

The CTO of LocalHalo was contacted by email on 26th March 2020 to enquire about the high crash rate. He acknowledged it on the same day and promised to reply ‘asap’, but only followed up a month later. In the interim LocalHalo also released another version of their app. The CTO believed the cause of the high crash rate was the Expo framework. He apologised for the delay; they had been very busy as a small team who had been working at maximum capacity.

In summary, the data was collected from five sources: 1) Google Play Console and Android Vitals, 2) Sentry’s weekly reports, 3) Sentry’s online service, 4) via email, from the CTO, and 5) through grey materials.

Analysis: The automated emails from Sentry provided a useful history over several years of the usual pattern of errors and failures that Sentry detected. The weekly reports were analysed, as were the online reports in Sentry and in Android Vitals.

Figure A.1 illustrates the distinct sections of a weekly report in Sentry using dotted rounded rectangles. Beacon-finding was used to identify four areas for further investigation:

1. Start and end dates: These identify the week covered in this report. These weekly reports started on a Monday and finished on the following Monday (8 days). However the graphs were for 7 days, from the first Monday until the following Sunday (7 days). The emails were received consistently on the end date, at around midday UK time.

2. Projects listed: In this example there is only 1 project, react-mobile, which appeared in every weekly report over 2 years. localhalo-webswit4 also appeared in many of these reports; as it was for a non-mobile project, that project’s results are not evaluated in this research. The
name of each project was a hyperlink to the respective project in Sentry; access was only available to pre-configured user accounts.

3. **Events by Project**: This is a bar chart for 7 contiguous dates, from Monday to Sunday. When both projects appeared in the report, the bars were stacked to show the cumulative amount of errors. This graph is very similar to the much smaller and less-detailed ‘Events Seen This Week’ graph in this weekly report.

4. **Daily events for the current month**: This is the last of three similarly-patterned calendar month reports with a single colour per calendar day in that month, organised in a row per week. These started on a Sunday and finished on the next Saturday. The choices of colours were not documented in the weekly emails or online; it can be inferred the strength of the orange colour is an indicator of volume of something.

As the weekly report had a hyperlink to the respective project in Sentry’s online dashboard, that link was used to drill-down into more granular online reports.

**Findings, and how they were validated**: Abnormal behaviour was reported that contradicted the typical daily patterns. The abnormal behaviour was split across two of the weekly reports provided by Sentry. The contents of 4 contiguous weeks of Sentry’s reports during March 2020 are shown in Figure A.2. They illustrate the lack of errors being reported by Sentry for the period of 17th to 25th March 2020. Additional research into bugs reported for the React Native Expo framework provided a correlating adverse condition that was probably causative.

These reports were compared to those from Android Vitals, which reported failures during this interval; these are shown in Figures 8.2 and 8.3, on pages 184 and 185 respectively.

With the exception of the period in March 2020, Android Vitals did not provide any reports for the LocalHalo app during the active case study. Conversely, Sentry reported errors for nearly every day over several years.

---

6: Access to Google Play Console and Android Vitals was available for approximately six months in 2020.

7: The nature of the weekly reports means distinguishing between zero errors for the mobile app and a small number of errors is not easy to do visually.
Several findings emerged from the characteristics of the Android Vitals reports and using Sentry. These include: 1) the benefits of combining tools, 2) third-party software challenges, 3) SDK crashes app, 4) inattention leads to entropy, 5) bug found using mobile analytics and tracked, and 6) weaknesses in dev practices.

Android Vitals was of little use when the cross-platform runtime worked as intended. Also, developers who use React Native need to incorporate in-app mobile analytics to obtain crash reporting and other insights into the behaviours of their app in the field and how it is being used. However, when the runtime failed, Android Vitals provided rapid feedback and sufficient information in the crash clusters to identify the cause of the failure. Sentry was a good product fit for their cross-platform app; however, this small startup was often too busy to check the reports – checking the reports was not integrated into their workflows. The reports from Android Vitals were not directly actionable, however they were sufficient to enable appropriate action to be taken through additional research of grey material. Android Vitals is a poor product fit for apps written in React Native (at least from two data points, i.e., from LocalHalo and from GTAF).

The findings from mobile analytics and the reported issue 5839 [269] were discussed with the CTO of LocalHalo, who accepted that the behaviours were plausible; he had not checked the reports during the period of the abnormal behaviours. Documentation from the Expo project on error handling [345] corroborates the design of automatic error handling and recovery when using Expo for React Native.

The monthly calendar graph provided an indication of some activity, and indicates clearly the gap where no data was received. There are flaws in the graphs, including a lack of legends or labels. Also, the calendar graphs for the previous two months are inconsistent from week to week and month to month. This indicates that there may be flaws in the tools and/or services provided by Sentry.

**Conclusions:** The LocalHalo case study provided unexpected insights into the behaviours of error handling in React Native apps that use

**Figure A.2:** Sentry: weekly emails for errors in LocalHalo in March 2020

[269]: Nieuwenhuis (2019), *Apps sometimes crash (call stack at BaseExperienceActivity$2.run)*

[345]: Expo (2023), *Error handling*
the popular Expo framework. As there was only very limited access to another React Native app’s mobile analytics, we only have tentative indications of patterns in the failures that the platform analytics (Android Vitals) are able to detect. Additional research into React Native and other cross-platform frameworks would be desirable to help determine if these findings can be generalised.

A.3 Connecting failures pre- and post-app launch

Google Play Console is able to match crash clusters found in production that were also found by their pre-launch testings. This example occurred for the core Kiwix Android app in November 2019. The key information is in the following extract from the report:

Crash cluster similar to a crash in pre-launch report for APK 180980

To resolve this issue, use the information available in pre-launch report in conjunction with details of this similar crash cluster.

Conversely, the reports in Android Vitals mentioned when that crash cluster had also been reported in the pre-launch report via the automated testing performed as part of that report. Figure A.4 includes two such examples where the cross-connection has been established, one for Play (start) Download, and the other for Pause Download.

Data collection: The data was collected during the Kiwix Case Study, reported in Section 5.6. Figure A.3 presents the captured information while manually drilling-down into an individual crash cluster.

Analysis: The contents of the two crash stack traces were compared and found to match the class and method names for each line in the stack trace. The line numbers do not match, which is consistent with investigating stack traces for Android code across Android releases, so unsurprising9. The pre-launch report provided more details, including a recording of the automated testing that reproduced the crash.

Findings, and how they were validated: The crash was triggered by the automated testing performed by Google automatically for a test release of the Kiwix Android app. Google also detected that this crash occurred in production on at least one end-user’s Android device, called a Tecno SPARK 3 running Android 9. We were unable to validate who the user was or what they experienced, as those details are not provided by Google. The finding was validated by watching the screen recording of Prelaunch testing in Google Play Console10.

Author’s Note A.3.1 Facilitating crash reproductions in industry

Were Google to provide details of the inputs that were performed when testing the app, for example as a script file, developers could try to reproduce the inputs subsequently.

Google does provide Android Monkey and the automated pre-launch...
A.3 Connecting failures pre- and post-app launch

Figure A.3: Example of a crash found pre- and post-launch for the Kiwix Android app

Figure A.4: Crash clusters that mention similar clusters in pre-launch reports: Kiwix, 2019 (repeated from Chapter 8 for ease of reading)

tests appeared to use it, or something similar. Android Monkey can generate a script that can be used subsequently to reproduce the inputs it performed, which is useful for improving repeatability of the automated tests for bugs that those steps trigger (some bugs depend on other factors and/or conditions).

During this research, we experimented with using Android Monkey to reproduce crashes reported in pre-launch reports. We also collaborated with a startup called Test1080 (now defunct) that generated test scripts written in Python that were also able to trigger crashes in the Kiwix app. Their approach also generated inputs, based on screen coordinates, a technique similar to that used in Prelaunch testing. Their work helped confirm that the approach was able to reproduce crashes that also occurred in production. Unfortunately, they did not respond after sharing their initial results and test scripts, so that approach was not explored further.

Conclusions: This example indicates that Google Play was able to correlate crashes that occurred both in production and during their automated Prelaunch testing. These correlations were only observed infrequently,
and Google does not appear to have published their findings regarding their matching of pre- and post-launch crashes.

Partly because of the potential adverse effects of trying deliberately to reproduce crashes of the apps in production, we did not actively pursue in-depth testing or evaluation of their reporting.

A.4 Google Play Console reports

Google Play Console incorporates Android Vitals, and collectively they were used and studied in every one of the app-centric case studies. This worked example focuses on their characteristics and capabilities over approximately 13 months from May 2019 until June 2020.

<table>
<thead>
<tr>
<th>Details of access to Google Play Console accounts</th>
</tr>
</thead>
<tbody>
<tr>
<td>At the start of this period I only had read-access to the reports for the Kiwix project. Later on during this period, I was granted read access to Google Play Console for the following projects: Catrobat with ongoing access, eduVPN for ≈ 3 months, GTAF for ≈ 12 months, and LocalHalo for ≈ 6 months. Subsequently, I also had direct-read access for the specific app in the commercial C1 case study for ≈ 9 months. I continue to have read access for the Catrobat and Kiwix project accounts.</td>
</tr>
<tr>
<td>The rest of the app-centric case studies provided copies of reports, so my access was indirect and limited to when they provided copies of their reports.</td>
</tr>
<tr>
<td>I also had and have: a) owner access to a personal Google Play Console developer account, and b) write access to an account shared with Joseph Reeve, which we used for several of the mini-experiments.</td>
</tr>
</tbody>
</table>

Data collection: The data consists of an extended screenshot of the dashboard report that Google Play Console displayed for Catrobat’s Pocket Code Android app. It was collected using Firefox for Mac’s ‘Save screenshot’ right-click menu. This is a single instance of over 50 screenshot images collected during the research; it is presented here as a typical example of the analysis described in this section.

Structure and content: In Figure A.5, the dashboard has 5 expandable sections (expansion is indicated in blue at the right of each section). For example, the top-most section ‘How are your KPIs performing?’ is expanded, as are the 3 middle sections – unlike the bottom-most section “Is your app’s size optimized”, which is not expanded. Every instance of this dashboard contained an even number of graph elements. The developer can toggle the expansion/contraction of each section. The discussion that follows will number the individual graphs left-to-right and top-to-bottom, with odd numbers on the left and even on the right, so the top left graph is 1 and the bottom right graph is 12.

1. New users acquired: 6.33K +1.67% vs previous period. The graph has a single vertical bar labeled (1,200) of the 4 in the graph, and 3 dates labeled (Sep 19, Sep 21, Sep 23). New users acquired are
Figure A.5: 1K crashes and ANRs in last 7 days Screenshot 2019-09-25 Dashboard - Pocket Code - Google Play Console
plotted in blue, and New users acquired (30 days rolling average) are plotted in light blue.

2. **Users lost**: 8.74K +3.97% vs previous period. The graph has a single vertical bar labeled (1,600) of the 4 in the graph, and 3 dates labeled (Sep 19, Sep 21, Sep 23). Users lost are plotted in blue, and Users lost (30 days rolling average) are plotted in light blue.

3. **Average rating**: 4.03 +7.17% vs previous period. The graph has a two vertical bars labeled (1, 5) of the 4 in the graph, and 3 dates labeled (Sep 17, Sep 19, Sep 21). The daily average rating is plotted in blue.

4. **Crashes & ANRs**: 1.00K +10.57% vs previous period. The graph has a single vertical bar labeled (200) of the 4 in the graph, and 3 dates labeled (Sep 19, Sep 21, Sep 23). Crashes are plotted in blue, and ANRs in yellow.

5. **New users acquired**: 6.33K +1.67% vs previous period. The graph has a single vertical bar labeled (1,200) of the 4 in the graph, and 3 dates labeled (Sep 19, Sep 21, Sep 23). New users acquired are plotted in dark blue, and New users acquired (30 days rolling average) are plotted in light blue.

6. **Users lost**: 8.74K +3.97% vs previous period. The graph has a single vertical bar labeled (1,600) of the 4 in the graph, and 3 dates labeled (Sep 19, Sep 21, Sep 23). Users lost are plotted in dark blue and Users lost (30 days rolling average) are plotted in light blue.

7. **New users acquired by country**: 35.87% new users acquired in Brazil 814 new users. A flat map of the world has colour highlighting where Brazil is the deepest shade of blue on the map, followed by Russia.

8. **Top countries**: 23.83% new users acquired in Russia 814 new users. An ordered horizontal bar chart with 5 blue rows is shown in descending order of new users. These are labeled on the left with the name of the country and on the right with a percentage. Here these values are: Brazil 36%, Russia 24%, Poland 5%, Ukraine 5%, Kazakhstan 2%

9. **ANR rate**: 0.28% 0.34% in previous 30 days. The graph has a single vertical bar labeled (1.00%) of the 4 in the graph, and 4 dates labeled (Aug 29, Sep 5, Sep 12, Sep 19). The daily ANR rate, titled ‘impacted sessions and related confidence interval’, is plotted in dark blue. A horizontal red line is also plotted on the graph for the ‘bad behavior threshold’.

10. **Crash rate**: 3.92% 4.15% in previous 30 days. The graph has a single vertical bar labeled (6.00%) of the 4 in the graph, and 4 dates labeled (Aug 29, Sep 5, Sep 12, Sep 19). The daily crash rate, titled ‘impacted sessions and related confidence interval’, is plotted in dark blue. A horizontal red line is also plotted on the graph for the ‘bad behavior threshold’.

11. **Average rating**: 4.03 +7.17% vs previous period. The graph has a two vertical bars labeled (1, 5) of the 4 in the graph, and 3 dates labeled (Sep 17, Sep 19, Sep 21). The daily average rating is plotted in blue.

12. **Ratings breakdown**: Most popular rating 66.67% 5* 66 ratings. An ordered horizontal bar chart with 5 rows is shown in descending order of star rating. These are labeled on the left with the star rating and on the right with a percentage; each has a distinct colour. Here
the values and colours are: 5* green 67%, 4* light green 8%, 3* orange 6%, 2* lighter red 2%, darker red 18%

Figure A.5 also includes a header section that incorporates a calendar dropdown, showing Last 7 days here, and a link to the public page for the current app. In the ‘How is your audience growth?’ grouping, there is a link to ‘VIEW ACQUISITION REPORT’ and a message regarding store listing experiments (a topic that’s outside the scope of this research). The final unexpanded section ‘is your app’s size optimized?’ includes a link to ‘VIEW APP SIZE REPORT’. It also has a message about 1 unimplemented size optimization and a link to ‘VIEW OPTIMIZATIONS’; this is followed with the App size over time 20.7 MB download size.

Analysis: This seemingly basic dashboard contains lots of information, especially if compared to other recordings of the dashboard for: other date ranges, this app on other dates, other apps for this developer account, apps for other developer accounts, and so on. Various salient perspectives on the contents follow. Note: in the interests of brevity, some minor analysis has been excluded, and some findings have been aggregated.

Beacon-finding, described in 4.3.3, identified dates and points on the various graphs, together with the textual data presented on each graph, as pertinent. Dates included the first and last dates in a graph, the granularity, and the relative nature of a date to the range selected in a report (where they can be selected) and by a report for predetermined date ranges.

This and other similar dashboard reports for Kiwix and Catrobat, in particular, were analysed using the sense-making methods described in 4.3.3 and sense-building. As an example of sense-making, Figure A.6 shows how hovering over each of the first and last points on the line on the graph shows the first and last date contained in this report. And by repeating a similar check of the starting date when drilling-down into the linked report: Statistics, the start date actually differs from the start date of the parent report!

In terms of sense-building, across-case comparison (covered in Chapter 4 in Section 4.3.4) found a consistent pattern the flaw illustrated in figure A.9 from the Kiwix case study which corroborates one of the findings where the second-ranked country is the headline for the bar chart of the top 5 countries graph. In Figure A.9 instance, China is the title, even though it’s the second-ranked country in terms of use users of the Japanese language version of the Wiki-Medicine app – despite Japan being overwhelmingly the top country in terms of new users. Note the mislabelling of the top countries report is corroborated in Figure A.10.

Drill-down into the graphs and reports linked from the respective graphs uncovered additional findings, including further differences in date ranges. These are illustrated in Figures A.6, A.7, and A.8. The date range in the dashboard report was from 8th September to 7th October 2019 (30 days including both days), while the linked statistics report was from 9th September to 7th October 2019 (29 days including both days).

The findings were presented to and jointly analysed with the Google Engineering team as per the ‘Ask the tool devs’ method, described in 4.3.5. They requested a report, which I provided, that categorised the
Figure A.6: Kiwix Chemistry and Physics Simulations app: Combined screenshots showing the first and last date for the 'last 30 days'.
findings. They validated the findings through discussions and email correspondence with the Product Manager of Google Play Console (who was acknowledged, with his agreement, in [2]).

The dashboard and other related reports were also sanity-checked using ‘ask the app devs’ method (see Section 4.3.5. Ask the app devs). One of the local app experiments (see Section 4.3.4. Local-app experiments) was used to determine if and when the dashboard would be populated for an Android app as it acquired the first 10 users. Further experimentation and evaluation were put on hold when Google refused to support more in-depth testing.

**Findings, and how they were validated:** The findings illustrate various flaws identified in Section 8.4.1. Flaws in the mobile analytics tools and/or services:

- **Dates:** The majority of the graphs are for the period selected in the header (last 7 days); however the two graphs for Android vitals are for 30 days, and the app size does not appear to have a period specified.

- **Duplicated graphs:** 3 graphs were duplicated in this dashboard: 1) new users acquired, 2) users lost, and 3) average rating. The duplication appears wasteful. The graphs often lacked information about the value represented by the lowest line on the vertical (Y) axis. The graph lines did display the value for each data point so the scale could be calculated; nonetheless, the value of the baseline needs to be determined by the reader rather than being easily available in the report.

- **Plausibility:** 8.74K users lost is more than the 6.33K new users acquired. If these figures refer to a userbase in common then the overall userbase has declined by 2.41K users in 7 days. No mention is made of returning users who were ‘lost’ who are now ‘found’ again, so there appears to be a possible gap in the overall picture of the userbase. Nonetheless, any app that loses more users than it gains would end up with close to no users at all.

- **Second-ranked country highlighted in top countries graph:** The top countries graph presents the results of the second-ranked country (Russia) rather than the first-ranked (Brazil), as per the graph immediately to the left/preceding this graph.

- **Consistency:** 1) The date ranges are not consistent for all the graphs in the dashboard report. Android Vitals graphs have only been seen with a 30-day range.[11] 2) The counts for users are not consistent across the 4 graphs grouped under audience growth. Are there 6.33K new users or 814? Or does 814 represent the new users in Brazil and/or in Russia?

- **Utility:** The Crashes & ANRs graph presents both data sets on a common chart. [How] can these be usefully compared and contrasted?

- **Freshness:** The Ratings graph is for a different date range (16 to 22 Sep 2019), two days earlier than the rest of the 7-day graphs (18 - 24 Sep 2019). The Ratings graph is stale compared to the others.

The dashboard reports were not considered to be particularly useful in 2019 and 2020 by the app developers involved in this research. Google revamped the dashboard in 2022; many new graphs were added and

---

[11]: Potentially some other users of Google Play Console are presented with other ranges so we cannot generalise beyond identifying that for every app and for every team and for every example the range has been 30-days for these graphs.
developers could now customise which graphs appeared. In May 2022, Google also launched various new services including the SDK index and an API to enable developers to integrate Android Vitals into their workflows [346]. These improvements corroborate the indications from the app developers that the dashboard was previously not particularly useful.

**Conclusions:** For this example, the primary and key oracle was the product and engineering team at Google who were responsible for Google Play Console and Android Vitals. They actively sought findings and updates generated from this research, which indicates that the research was contributing valid findings deemed valuable to Google.

### A.5 Is Android Vitals Alive?

For a period of at least 10 (ten) days in September 2019, the graphs and reports for Android Vitals were not updated, so the data was stale.

**Data collection:** The data was collected by visiting Google Play Console interactively on an ongoing basis, including during September 2019, and taking screenshots of the dashboard. Two Google Play accounts were checked (Kiwix and Catrobat, *i.e.*, those that were available to the researcher at that time) and similar issues were observed for all the apps that had Android Vitals data in both developer accounts. Figures A.10 and A.11 are two examples of the stale data in the respective dashboard reports for the main Kiwix app and for Pocket Code.

Figure A.13 has been cropped and annotated with three arrows to highlight: 1) this report was obtained on 16th September 2019; 2) it is for the last 7 days; and 3) the last date in the report is 3rd September 2019.

**Analysis:** Three techniques were used in particular: 1) drill-down to compare the headline graphs presented in the dashboard report with those presented in the Android Vitals section of Google Play Console. Figure A.12 combines extracts of the crash rates for the main Kiwix...
A.5 Is Android Vitals Alive?

Figure A.10: Google Play Console dashboard: Android Vitals data is stale - Kiwix

Figure A.11: Google Play Console dashboard: Android Vitals data is stale - Pocket Code
Figure A.12: Composite image: Android Vitals for two of the Kiwix apps showing gaps in data

Figure A.13: Kiwix: Android Vitals: Last 7 days report for 16th Sept 2019

Android app and for the Chemistry and Physics Simulations app. 2) Across-case comparisons for the various apps that had Android Vitals reports both within and across developer accounts. 3) Ask the tool devs - The Google engineering team was asked and the issues were documented in the report they requested. They did not provide any explanation of the gaps in the reports.

Note: these reports also contained other gaps in the graph. These gaps occurred intermittently on a quite frequent basis on a per-app basis, rather than across multiple apps and developer accounts. They were analysed using across-case comparison. Searches for any status reporting were performed within Google Play Console, the relevant online help pages, and online generally. No relevant information was found.

Findings, and how they were validated: Both Google Play accounts had similar gaps in the graphs across the apps that had Android Vitals reports (the volume of activities was too low for Google to provide graphs for some of the apps on each developer account).

The findings were validated implicitly by reporting them to Google and in subsequent discussions with their product and engineering
teams. ‘Implicitly’, as they chose not to answer the points raised directly; nonetheless, they did not challenge the reports of these flaws, whereas they had challenged other flaws previously (and those challenges were upheld).

**Conclusions:** Google Play Console appeared to have at least one problem which led to gaps in Android Vitals reports. Some gaps aligned across several apps and developer accounts. Google did not provide status updates for Android Vitals; they did and continue to do so for other online services (e.g., for Google Workspace Status Dashboard and for Google Analytics - 5 year history.
Thematic Analysis

The case studies provided a plethora of observations; these needed to be analysed to establish if common themes emerged. In terms of the six perspectives that had emerged from the case studies, three core groupings were used: for processes, for artefacts, and for analytics tools with the aim of developing the foundations for three chapters of this thesis. Each chapter then discusses the use and improvement aspects.

There is a wide range of software tools available for thematic analysis. A spreadsheet was the chosen tool, partly to facilitate sharing and collaboration.

Use of Google Spreadsheets to map content to chapters Google’s service was chosen, as it facilitates a wide range of collaboration without participants needing to pay for software licenses. It also provides useful spreadsheet formulae such as FLATTEN\(^1\) not available in Microsoft Excel at the time of writing.

\(^1\) support.google.com/docs/answer/10307761

Figure B.1: Thematic Analysis Spreadsheet Design
B.1 Spreadsheet design and use

The spreadsheet was designed to take advantage of core spreadsheet features: worksheets for the content, the calculations, the analysis and the mapping, named ranges to identify datasets, and spreadsheet formulae to perform calculations. To make the data easier to process, compound-names and full-stops (periods) were used in the theme data cells. And finally, there are two levels of thematic groupings, L2 and L1, where L2 combines L1 themes for reporting purposes.

An artefact can include several themes; where this occurs, the themes are listed in the same cell and separated with commas.

B.1.1 Worksheets

The spreadsheet contains three distinct types of worksheet. The first lists artefacts and their primary source(s). The second type includes areas for calculations, together with descriptions of all of the respective granular level one (L1) and level two (L2) themes. And the third type is used to identify commonalities for the artefacts that share an L1 theme, their primary focus (research, practice, or both), and a mapping to a section in the respective chapter of this thesis. These three distinct types are illustrated in Figure B.1.

B.1.2 Compound names and full-stops

Themes incorporate hyphens to join individual words into a compound term, and this term can be processed relatively easily with spreadsheet formulae while also being palatable in \( \LaTeX \). Examples of themes include: use-of-automated-tests and actionable-reports.

The full-stop character was used in place of an empty cell to enable auto-complete to work down columns that would otherwise have blank cells in some of the row entries. These full-stop characters are accounted for throughout the spreadsheet so that they are not counted.

B.1.3 Spreadsheet “named ranges”

Use of Named Ranges made datasets easier to visualise and identify within the spreadsheet, compared to using hard-coded cell identifiers in spreadsheet formulae. There are individual named ranges for:

- process, artefact, and analytics-tool topics,
- process, artefact, and analytics-tool themes,
- areas used as working storage for formula calculations.

The named ranges also use compound names; however, rather than using hyphens, the first letter of each word is capitalised, for example ProcessThemes.
B.1 Spreadsheet design and use

B.1.4 Spreadsheet formulae

Standard, but not necessarily commonly-used, formulae are used to perform calculations, data expansions, aggregation, and analysis. These are all valid in Google Sheets; some formulae may need revising if the spreadsheet is used Microsoft Excel.

Examples of the spreadsheet formulae used in the thematic analysis include:

- **Unpack the comma-separated list of themes using the relevant NamedRange, e.g.**
  
  '=(arrayformula(TRIM(iferror(split(Content!ProcessThemes,","),""))))', enter a blank string if none found.

- **=COUNTIF(AnalyticsToolsL1Expansions, D2); Count of L1s.**

- **=RANK(B2, AnalyticsToolsL1Counts); Rank of L1 theme.**

- **Thematic analysis**
  
  - =(COUNTUNIQUE(arrayformula(TRIM(split(ProcessThemes,"")))))-1 provides a count of all the unique L1 terms found in the themes for processes.
  - =COUNTIF(ProcessThemes, "+improve_engineering_practices*) provides an example of searching for a specific theme amongst the many.
  - =ARRAYFORMULA(FILTER(ProcessThemes, REGEXMATCH(ProcessThemes, ".*improve_engineering_practices.*"))) finds the other terms combined with the search term.

- **="Source project|tool ("&COUNTA(SourceProjectOrTool)&")"; a count of the findings recorded in the spreadsheet.**

- **="Processes ("&COUNTA(D5:D1000)&")", ="Artefacts ("&COUNTA(F5:F1000)&")", ="Analytics Tools ("&COUNTA(H5:H1000)&")"; added the count of items to each of the column headings in the Content worksheet.**

- **=sort(UNIQUE(F3:F37))** generates an ordered list of the distinct themes from the overall set of L1 themes for Artefacts. Similar formulae exist for the Processes and Tools perspectives.

- **=RANK(G3, AnalyticsToolsL1Counts) Ranks the frequency of the L1 themes for any of the three chapters the present the findings.**

- **Internal Validation Checks**
  
  - =NOT(AND(ISBLANK(F5), ISBLANK(H5), ISBLANK(J5))); a basic check to help check at least one L1 theme has been provided for each finding. Used on the content worksheet.

  - Additionally data validation was used to check the L1 themes that were entered in the Content worksheet were found in the respective named range e.g. =L1ThemesProcessUse is the named range for the L1 themes in the Processes column (Column D). Some Cells contain several L1 themes, for these the data validation rule added a warning indicator and accepted the multiple values. Figure B.2 shows the data validation rules.

---

**Figure B.2:** Example of the data validation rules in the Google Sheet for Thematic Analysis
B.1.5 Inclusion criteria for a findings chapter

The top-ranked L1 themes were selected for each of the three findings chapters: Chapters 6, 7, and 8, as they have the strongest base of evidence. These were augmented by key topics, even if they did not rank as highly as the top $n$; and additional L1 themes were also included within the dissertation when it was germane to do so, for instance to support a point or provide additional evidence.

B.2 Ishikawa Diagrams

The Ishikawa (fishbone) diagrams were created in online software called miro using a free account. As noted in Chapter 4 on page 90 they were reviewed with several academic colleagues who cross-checked the outputs of the thematic analysis and the groupings of the themes.
Figure B.3: Illustration of the figure that illustrates the mapping of L1 and L2 themes for Processes; also known as Analytics in Use.
Findings:
in artefacts when mobile analytics used to improve mobile apps

Themes for Apps and their Artefacts (g) perspectives

- App-facing topics
- Crash rates
- Improvements
- Bug reproducibility
- Build process & results
- Build scripts
- App crashes
- SDK
- Mobile analytics
- Testing using mobile analytics
- Efficacy of addressing bugs
- Bug reproducibility
- Build process & results
- Build scripts
- Bug crashes
- SDK
- Mobile analytics
- Testing using mobile analytics
- Efficacy of addressing bugs
- Bug reproducibility
- Build process & results
- Build scripts
- Bug crashes
- SDK
- Mobile analytics
- Testing using mobile analytics
- Efficacy of addressing bugs
- Bug reproducibility
- Build process & results
- Build scripts
- Bug crashes
- SDK
- Mobile analytics
- Testing using mobile analytics
- Efficacy of addressing bugs
- Bug reproducibility
- Build process & results
- Build scripts
- Bug crashes
- SDK
- Mobile analytics
- Testing using mobile analytics
- Efficacy of addressing bugs
- Bug reproducibility
- Build process & results
- Build scripts
- Bug crashes
- SDK
- Mobile analytics
- Testing using mobile analytics
- Efficacy of addressing bugs
- Bug reproducibility
- Build process & results
- Build scripts
- Bug crashes
- SDK
- Mobile analytics
- Testing using mobile analytics
- Efficacy of addressing bugs
- Bug reproducibility
- Build process & results
- Build scripts
- Bug crashes
- SDK
- Mobile analytics
- Testing using mobile analytics
- Efficacy of addressing bugs
- Bug reproducibility
- Build process & results
- Build scripts
- Bug crashes
- SDK
- Mobile analytics
- Testing using mobile analytics
- Efficacy of addressing bugs
- Bug reproducibility
- Build process & results
- Build scripts
- Bug crashes
- SDK
- Mobile analytics
- Testing using mobile analytics
- Efficacy of addressing bugs
- Bug reproducibility
- Build process & results
- Build scripts
- Bug crashes
- SDK
- Mobile analytics
- Testing using mobile analytics
- Efficacy of addressing bugs
- Bug reproducibility
- Build process & results
- Build scripts
- Bug crashes
- SDK
- Mobile analytics
- Testing using mobile analytics
- Efficacy of addressing bugs
- Bug reproducibility
- Build process & results
- Build scripts
- Bug crashes
- SDK
- Mobile analytics
- Testing using mobile analytics
- Efficacy of addressing bugs
- Bug reproducibility
- Build process & results
- Build scripts
- Bug crashes
- SDK
- Mobile analytics
- Testing using mobile analytics
- Efficacy of addressing bugs
- Bug reproducibility
- Build process & results
- Build scripts
- Bug crashes
- SDK
- Mobile analytics
- Testing using mobile analytics
- Efficacy of addressing bugs
- Bug reproducibility
- Build process & results
- Build scripts
- Bug crashes
Findings: in Mobile Analytics tools used to improve mobile apps

Utility

Design

Dependability

Fit for purpose

SDK-design

UX-design

Product fit

Actionable reports

Integration into workflows

Metadata

Efficacy of tool

Benefits of combining tools

Engineering challenges

Bug-localisation

Trust-worthiness

Testability

Link-rot

Flaws

Fit for purpose

Return On Investment

Ethical considerations

Integration into workflows

Product fit

Fit for purpose

Platform-oriented

Dev-oriented

e.g. emails from mobile analytics tools

Figure B.5: Illustration of the figure that illustrates the mapping of L1 and L2 themes for Tools; also known as Tools and their Artefacts.

Themes for Analytics Tools and their Artefacts perspectives last revised 29 May 2023 (g)
This appendix presents additional illustrative material for several of the micro-experiments presented in Chapter 5 in Section 5.9.1.

C.1 Adding meta-data to crash logs

Listing 9 is an extract of source code we wrote to augment a stack trace with meta-information about the Android device and the operating system. It was implemented in one of the mini-experiments called AndroidCrashDummy, which is opensourced and available on GitHub github.com/ISNIT0/AndroidCrashDummy.

A similar incomplete implementation was found in grey data of an uncaught exception handler, published on Stack Overflow, that was designed to report other meta-information via Crashlytics [270].
Listing 9: Extract of java code that adds meta-data to crash stack traces to an Android app’s codebase

```java
@override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    final String deviceInfo = "==Device:[" + Build.DEVICE + "]
    + ", Model:[" + Build.MODEL + "]
    + ", Manufacturer:[" + Build.MANUFACTURER + "]
    + ", Time:[" + Build.TIME + "]
    + ", Android Version:[" + Build.VERSION.RELEASE + "]==";
    final Thread.UncaughtExceptionHandler systemHandler = Thread.getDefaultUncaughtExceptionHandler();
    Thread.setDefaultUncaughtExceptionHandler(
        new Thread.UncaughtExceptionHandler() {
            @Override
            public void uncaughtException(Thread t, Throwable e) {
                StackTraceElement[] trace = e.getStackTrace();
                StackTraceElement[] newTrace = new StackTraceElement[trace.length + 1];
                System.arraycopy(trace, 0, newTrace, 1, trace.length);
                newTrace[0] = new StackTraceElement("==Device Info==", deviceInfo, ",", -3);
                e.setStackTrace(newTrace);
                systemHandler.uncaughtException(t, e);
            }
        });
}
```

C.2 Experiences with a test Android app: Travel Europe

The author co-created a small functional test app, called Travel Europe, with Joe Reeve. It is available in Google Play for pre-release testing (it has not been published yet). The app incorporates Microsoft’s App Center Analytics and content known as Travel Europe that was collated automatically by the Kiwix project and made freely available for use and re-use.

We created the Android test app in Kotlin. Initially, it had at least one known flaw (in release 6) that caused the app to crash with an unhandled exception. Several test releases were uploaded to the app store over several weeks.

We wanted to track whether Google Play Console detects the installation, activity, and any crashes that occur.

C.2.1 From Zero to Ten

For our test app we configured 10 additional valid Google accounts and invited each of these to be testers of the new as yet unreleased app. These accounts belong to a domain owned by one of the authors; otherwise they are standard Google accounts as far as we know. Each of these accounts was assigned to a different physical Android device.

There was a per-account opt-in to be a tester for the app (using a convoluted process) before the app could be downloaded.¹ Version 6 was installed and run on each of the devices. After the app had crashed on each of the devices, the mobile analytics reports were checked.

1: This would not be necessary for apps available as a general release in Google Play.
An updated release of the app (release 7) was created that fixed the crash, and it was uploaded to Google Play as another test release. This was installed and run on each of the devices.

### C.2.2 Pre-launch report automated testing

For release 6 of the test app, crashes were found on 10 of the 12 devices used by the automated monkey testing (which uses Google Android’s Robo automated test tool on their Firebase Test Lab’s physical devices). The test runs for the two crash-free devices did not happen to click on content that triggered the exception.

### C.2.3 Google Play Console and Android Vitals reporting

Initially, Google Play Console and Android Vitals included usage and errors for the app. These mysteriously stopped a few days later, and none of the ramp up to 10+ users has been recorded. As activity is shown for most days for most of the apps to which the authors have access, it is not clear whether the omission is by design, accident, or a glitch in the processing system.

### C.2.4 Microsoft App Center reporting

Microsoft’s App Center Analytics seems to track activity within a minute of it occurring. It recorded one active user in the United States, seven in the United Kingdom and 21 Unknown. Some of the ‘users’ may be those in Firebase Test Lab (used by Google for the automated pre-launch testing); however we were not able to determine the causes for the 21 unknown locations (the app does not explicitly ask for location permissions on behalf of the Microsoft App Center library incorporated in the app). It also counted one device (a HTC Desire 510) twice, perhaps as it had both debug developer builds and the app from the app store installed at different times?

App Center limits various details to the top 4 items, for instance the ‘Top devices’ and ‘OS Distribution’; this lack of detail is unhelpful. In contrast, Google Play Console sometimes provides many more items; the exact amount depends on several factors the author is still investigating.

Microsoft App Center did not detect crashes that happened in August 2019, the last instance they detected was 3 weeks earlier in July 2019, see Figure C.2.

### C.2.5 Conclusions

This mini-experiment provided insights into using in-app analytics during pre-release stages of app development, where some usage was recorded by Microsoft App Center. Disturbingly some of the crashes were not reported in either Microsoft App Center or Google Play Console with Android Vitals. Had none of the crashes appeared in Google Play Console, then one option was that they do not report on pre-release
Figure C.1: Microsoft App Center user growth for Travel Europe

![Graph showing user growth](image)

Figure C.2: Microsoft App Center crash reported for the Travel Europe app

![Crash report](image)

Apps; however the existence of some crashes indicates this hypothesis is incorrect.
This research is not intended to live in a vacuum! One of the aims throughout my research has been to contribute potentially-useful software and materials and make them freely available as opensource software under a permissive license for whoever wishes to use and improve them. This chapter itemises various projects on GitHub to which I contributed. Many of the projects I instigated, and I often co-developed them with Joseph Reeve, a friend and part-time colleague on an occasional basis. He is a prolific, fast, and responsive software developer (see github.com/isnit0 for his activities on GitHub), and he enabled me to expand the research significantly rather than relying on my limited software development skills.

D.1 Co-developed GitHub projects

- **Android Crash Dummy**  
github.com/ISNIT0/AndroidCrashDummy  
A small Android app to test logging and exceptions.

- **Android Log Assert**  
github.com/ISNIT0/AndroidLogAssert  
A library to facilitate and test log messages for Android apps. This project enables validation that the expected logs have the expected contents, as per Section 2.1.2. Communications paths and data flows.

- **Android Monkey Test with Login**  
github.com/commercetest/android-monkey-test-with-login  
An experiment to help increase the effectiveness of Android Monkey when it needs to interact with an Android app that uses a login page (since monkeys seldom enter a valid username and password. (Android Monkey is a very popular test automation tool used both by developers and researchers.)

- **Android Stability Analysis**  
github.com/commercetest/android-stability-analysis  
A utility script to pattern match clusters of errors reported in Android Vitals. More details are available in Section D.1.1.

- **(Android) Vitals Scraper**  
github.com/commercetest/vitals-scraper  
one of the major contributions of my research. This is described in Section D.4. Released as an npm package npmjs.com/package/vitals-scraper. The npm site reports several downloads per week, on average.

- **IDot**  
The source code is available on request at github.com/commercetest/IDot/; it currently includes some private configuration details so is not yet opensourced. It is based on AndroidCrashDummy and intended to facilitate testing and evaluation of mobile analytics services.
LogSearcher [GitHub](https://github.com/ISNIT0/log-searcher) A tool for searching Android codebases and analysing usage of “Log.*”. Includes a discussion in the README on the rich potential of designing and using logging.

Log Complexity Analysis
[GitHub](https://github.com/ISNIT0/log-complexity-comparison) An experimental early-stage project to try and identify complex code that lacks logging.

Logcat filter and analysis tool
[GitHub](https://github.com/ISNIT0/logcat-filter) A utility that runs on a developer’s computer to filter and analyse log messages for specified Android apps. Generates a JSON format output on request to facilitate additional processing.

Zipternet [GitHub](https://github.com/ISNIT0/zipternet) A small Android application generator, written in Kotlin, to provide a basic app with sufficient functionality to be potentially realistic and useful. It is similar in concept to the custom apps created and maintained by the Kiwix Android project team.

### D.1.1 Android Stability Analysis

Android Vitals does not group clusters completely which leads to flaws in their rankings of the errors. This scripts helps teams identify common clusters, the totals for these clusters can then be recalculated. The recalculated totals then enable a corrected league table to be produced. Teams can then choose the order to address reported errors based on the corrected ranking rather than the flawed ranking Google generates.

In August 2019 we started another opensource project [Android-Stability-Analysis](https://github.com/ISNIT0/android-stability-analysis) to enable scattered crash clusters reported in Android Vitals to be regrouped. Currently, the analysis tool simply matches reported crash clusters with a series of lines that should be common to related crashes, see Listing D.1 for an example.

#### Listing D.1: Example of lines to match

```plaintext
at org.kiwix.kiwixmobile.downloader.
DownloadService.pauseDownload(DownloadService.java:266)
at org.kiwix.kiwixmobile.downloader.
DownloadFragment$DownloadAdapter.
setPlayState(DownloadFragment.java:227)
at org.kiwix.kiwixmobile.downloader.
DownloadFragment$DownloadAdapter.
lambda$getView$5(DownloadFragment.java:286)
```

### D.2 Solo projects

Android analytics testing
[GitHub](https://github.com/commercetest/android-analytics-testing) Notes to help prepare for testing of analytics for Android devices. The notes may also suit testing of analytics on other platforms.
D.3 Contributions to external projects

Contributions to external projects that emerged from this research include:

- **Analytics Testing [using] Puppeteer**
  [github.com/dumkydewilde/analytics-testing-puppeteer](https://github.com/dumkydewilde/analytics-testing-puppeteer)
  Simply created a small README for the project. The project automates testing of Google Web Analytics.

- **EduVPN**
  [github.com/eduvpn/android](https://github.com/eduvpn/android)
  I created a proof-of-concept Continuous Build for the project [github.com/commercetest/android](https://github.com/commercetest/android), however the underlying projects were particularly complicated to build and problematic so the work has stalled pending decisions on the direction for the overall code-base.

- **Pocket Code (Android)**
  [github.com/Catrobat/Catroid](https://github.com/Catrobat/Catroid)
  I work with the project team, including the developers, to help them understand the principles and tools for using analytics as part of their development and bug investigation practices. I have also been involved in helping instigate and design several small projects in parallel which are not on GitHub.

- **Pocket Code (iOS)**
  [github.com/Catrobat/Catty](https://github.com/Catrobat/Catty)
  Worked with the lead developer to design and implement crash and mobile analytics using Firebase.

- **Kiwix Android**
  [github.com/kiwix/kiwix-android/](https://github.com/kiwix/kiwix-android/)
  The first of the case studies. I have been involved with this project since 2014 with a focus on software quality including testing. It formed one of the in depth app-centric case studies in this research.

- **Release Engineering for Mobile Apps**
  [shonan-releng-mobile/shonan-releng-mobile](https://github.com/shonan-releng-mobile/shonan-releng-mobile)
  Contributed material on an ongoing basis for international collaborative research on this topic.

- **Sentry documentation**
  [github.com/getsentry/sentry-docs/pull/4183](https://github.com/getsentry/sentry-docs/pull/4183)
  Improving the documentation of the use of their API was one of the micro-experiments
- to assess the ease of contributing to an opensource mobile analytics product.

- PostHog documentation
  
  [260]: Google Inc. (2020), Share usage & diagnostics information with Google
  
  google.com/PostHog/posthog.com/pull/1844, and
  
  google.com/PostHog/posthog.com/pull/1843

- mbox-to-csv
  
  jarrodparkes/mbox-to-csv/pull/21 added the step to install the dependencies. In this research mbox-to-csv helped facilitate the analysis of automated emails received from Sentry and Fabric Crashlytics.

D.4 Software we developed for Google Play Console

Developers are only able to read Android Vitals reports in real-time in the user interface Google provides in Google Play Console. This limits their ability to perform trend analysis beyond the period Google allots. Also, they are restricted to taking screenshots, etc. to save pertinent information. There are several limits: a hard limit of 60 (sixty) days, and seemingly limits based on volumes of crash and ANR data.

Our contributions enable developers to preserve various relevant reports and also individual crash and ANR trace data by using our Vitals Scraper software. They can then also use our software to group incompletely-grouped crash clusters, which enables them to improve the prioritisation of their work to investigate and improve quality aspects of their apps.

The developers can also choose to make the reports and data available to others, for instance for analysis and research. Our software is extensible and freely available as permissively-licensed opensource software tools.

D.4.1 Background

Google Play Console is a developer’s view of the Google Play app store and includes data and reports for the apps to which the person has access. Typically, developers have access to data for apps they develop and/or support.

At least some of the data is collected from users who opt-in to share their usage and diagnostics data with Google, and Google – in turn – shares some of this data with developers [260].

Google Play Console provides various summary data as monthly reports in a downloadable format, and these files are available historically, so the data can be downloaded for previous periods.
D.4.2 Summary Data is available for download

The monthly summary reports are available to download in Google Play Console\(^1\) either interactively or using a command line tool gsutil\(^2\). The reports include crashes and ANRs, ratings, reviews, installations, and financial information. From what we can tell, the data is available from when it was first generated by Google Play Console.

Note: Google used to provide summaries of crash and ANR reports until May 2018 [80] before they removed this facility. They also removed the ability to download these reports for prior periods, which left a gap in terms of being able to analyse either of these quality issues.

[80]: Google Inc. (2020), Download & export monthly reports
Bibliography

Here are the references in citation order.


[248] Best way to setup Release/Debug Firebase Analytics environment. url: https://groups.google.com/g/firebase-talk/c/3k3ZPnexF4s (cited on page 169).


[325] Ian Jonson and Sarah Neville. ‘NHS breaks up £400mn data contract in response to privacy concerns’. 2022. url: https://www.ft.com/content/a9228ce5-8662-474a-9b77-ca563cc5080a (cited on page 225).


Glossary and Acronyms

A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W

A
ADB
  Android Debug Bridge. 161
ADM
  Algorithmic decision-making system. (Source algorithmtips.org/2019/01/22/welcome/).
Analytics artefacts
  Analytics artefacts include published and unpublished outputs. Published outputs include data files and reports online and/or available to download. They also include results from calling APIs, and automated emails generated by analytics tools. Unpublished outputs including screen captures, screen-scraping and parsing.
Analytics system
  Analytics systems subsume the tool and provide a service. They generally include persistent storage of the collected data, they may also persist analysis and/or reports. Persistency may be for a finite period, and that may vary depending on pricing, data volumes, and so on..
Android developer account
  For the purposes of this research we use Google’s concept of a Google Play Developer Account, described variously in [348, 349]. Developers need to register for an account, pay a one-time fee, and agree to abide by various policies, terms, and conditions. Someone who has a developer account can invite other people to share aspects of their account.
Android vitals
ANR
  Abbreviation for Application Not Responding. A term originated by Google to identify when an Android app stops responding to users for over 5 seconds [236]. v, 8, 25, 38, 39, 55, 57, 60, 116, 118, 123, 150, 166–168, 183, 184, 188, 194, 197, 204, 210, 211, 223, 243, 254, 278, 279, 303, 305
API
  An Application Programming Interface (API) is a particular set of rules and specifications that a software program can follow to access and make use of the services and resources provided by another particular software program that implements that API. 16, 48, 57, 127, 135, 160, 175, 176, 182, 183, 186, 191, 194, 195, 212, 214, 229, 230, 232, 309
APK
  Android Package [Kit]. 53, 54
App
  A common abbreviation for a software application installed on mobile smartphone-based devices.
Application binary
  The file(s) that contain the compiled source code and associated libraries and resources needed to run the app on a device. 136, 158

B
Bad Behavior Threshold
  Google Play defines two bad behaviour thresholds that are used within the Google Play system to measure the crash rate and the ANR rate. The threshold for crashes is 1.09% and 0.47% for Gsplanr rate. 133, 134, 153
BDD
  Behaviour-Driven Development is intended to close the gap between business people and technical people, cucumber.io/docs/bdd/. 63
Breadcrumbs
Breadcrumbs are a series of small data messages written via the mobile analytics SDK and intended to "Breadcrumbs show you events that lead to errors". Source: https://sentry.io/features/breadcrumbs/.

Bus factor
Bus factor, aptly described as the "number of key developers who would need to be incapacitated, i.e. hit by a bus, to make a project unable to proceed" Cosentino, Izquierdo, and Cabot [350, p. 499].

C
Catrobat
Catrobat is a visual programming language and set of creativity tools for smartphones, tablets, and mobile browsers. Catrobat and the software developed by the Catrobat team are inspired by the programming system developed by the Lifelong Kindergarten Group at the MIT Media Lab. Catrobat — catrobat.org/ and developer.catrobat.org/ — is an independent non-profit project creating free open source software (FOSS) under AGPL and CC-BY-SA licenses. (Extracts from play.google.com/store/apps/details?id=org.catrobat.catroid., 309

CB
Continuous Build. 118,

CER
Microsoft’s Corporate Error Reporting service which includes a specific protocol, also known as CER.

Chrome.apk
Google’s Chrome browser appears on Google Android devices as a binary file called Chrome.apk, developers can choose to integrate it into their Android app to provide web browser capabilities and many developers have chosen to do so. 148

CI
Continuous Integration. 118,

Content Provider
‘Content providers are Android’s central mechanism that enables you to access data of other applications – mostly information stored in databases or flat files.’ Creating Content Providers. They are one of the fundamental architectural choices from the outset. Android’s online guide is online at developer.android.com/guide/topics/providers/content-providers . 182

CPU
Central Processing Unit.

Crash analytics
Analysis of application crashes collected automatically by software. Identify groupings and patterns in the crashes to provide developers with the opportunity to debug and fix them without needing to spend time reproducing the problem. (Paraphrased from [352]). v, 35, 48, 57, 139, 160, 175

Crash cluster
A grouping of crashes deemed to be effectively the same crash that occurred at least once. Often statistics and other information is provided in the crash cluster. Crash clusters are a subset of Failure clusters. 153, 192, 243, 244, 305

CSV
Comma, separated values. 194, 196

CTO
Chief Technology Officer. The most senior person who is responsible for the technology used and developed by an organisation, often at board level of a company or organisation. 103, 105, 133, 135, 138, 223, 239, 249

D
Data dynamics
The study of data moving within, through, between, and across computers including who has access to see some or all of the data, the contents of the data and the information that can be gleaned from it, etc. The concept is introduced from a testing of systems perspective in [6]. c.f. also: some concepts from physics and signals [353]. Note: data dynamics is not being used in terms of modifications to data
(see [354] and/or [355]), although those sorts of modifications may also occur in some circumstances e.g. to revise analytics data that’s been collected and forwarded.

**Developer / development team**
Both these terms are shorthand for people who are actively involved in the development of software. The work of developing software includes working directly with source code and working with other aspects of the project including design, testing, analysis, product and project management, UX, and so on. In this research these two terms are often used interchangeably where the choice aims to make the thesis more readable. Developers can have many roles and often do, and teams may include developers who work on more than one project or app. A development team includes multiple participants where communication increases in importance in terms of affecting the end results and in the human relationships.

**Development artefacts**
Development artefacts include the app binary, the source code for the app (including any source code incorporated into the app which is also developed and maintained by the app developers), build scripts, tests, documentation, work schedules, bug tracking systems, etc. Note: Third-party source code is generally excluded unless there is a clear and direct connection to the app, for instance through a bug report and/or a pull request created by the app developers. 71, 73

**Digital Feedback**
Here used to identify feedback from running software, generated automatically. The feedback is structured and the structure generally consistent.

**DSN**
Data Source Name. 159

**E**

**Error**
“An error is a system state that may cause a failure.” [356], based on [62].

**ESSE**
Empirical Studies of Software Engineering [193, p.1171]. 74,

**EULA**
End User License Agreement.

**Event demographics**
“Percentage of events triggered by each age group and gender.” (Source: Google Firebase Analytics tooltip).

**Exception**
Here, a programming language concept drawn from the Java programming language, that “In Java “an event that occurs during the execution of a program that disrupts the normal flow of instructions” is called an exception. This is generally an unexpected or unwanted event which can occur either at compile-time or run-time in application code. Java exceptions can be of several types and all exception types are organized in a fundamental hierarchy.” [357]. 9

**Exodus Privacy project**
An online privacy audit platform for Android applications (https://reports.exodus-privacy.eu.org/en/). 86

**Explicit feedback**
Feedback provided explicitly, for example, in the form of reviews and comments [34].

**F**

**Failure**
“A failure is an event that occurs when delivered service deviates from correct service.” [356], based on [62].

**Failure cluster**
A grouping of failures (Crash clusters or ANRs) deemed to be effectively the same failure that occurred at least once in the timescales of the report. Often statistics and other information is provided in the failure cluster. 192, 304

**Failure repository**
A central location in which data pertaining to failures of software is stored and managed. Notes: Definition based on a service, Oxford Languages, jointly provided by Google and Oxford University
Press. Mentioned but not defined in various sources including: [34], and exemplified in [358].

Fault
“A fault is the cause of an error in the system.” [356], based on [62].

F-Droid
A significant, yet minor, app store that hosts Android apps. All the apps are open source projects. The apps must not contain any proprietary, closed-source code. In practice this means they do not contain any in-app mobile analytics. f-droid.org/ . 66

FIM
Fault Inducing Module. 56

FOSS
Free and Open Source Software. 81, 82

FunDex
A score devised by Hewlett-Packard (HP) as part of their AppPulse Mobile product offering. “The score starts at 100, but drops with each problem the app has ...” [359]. It combines scores for UI performance, stability, and resource usage.

G
GDPR
General Data Protection Regulation (https://gdpr-info.eu/).

GitHub
The world’s most popular home for source code repositories. It provides private and public repositories. Much research into Android source code uses public repositories on About GitHub. 140, 144

GitLab
A popular home for source code repositories. It provides private and public repositories, see About GitLab for more details. 144

Google Play Console
A Web-based application which is the primary user interface for developers who release Android apps in the Google Play Store. Google also provides mobile apps that provide a subset of the capabilities of Google Play Console.

Grey material
Grey material includes grey data and grey literature. 71

GTAF
Greentech Apps Foundation, https://gtaf.org/, one of the case studies in this research. xix, 92, 100, 101, 133, 135, 138, 142, 143, 149, 150, 183, 234, 247, 249, 252

GUI
The Graphical User Interface (GUI) is a visible user interface, a touchscreen, on a mobile device. 104, 140

HTTP
HyperText Transfer Protocol. 56, 176

I
iArtefacts
Improve development Artefacts pertaining to the use of mobile analytics. One of the six perspectives identified in this research. 157

ICSE
International Conference on Software Engineering. 6

IDE
Integrated Development Environment. 42
IDO\text{\textscript{T}}
IDO\text{\textscript{T}} was a short name we created quickly for an experimental Android app written in Kotlin. The letters loosely represent: Iteratively Demo and Test Analytics as the aim was to demo and/or test Iteratively’s SDK’s capabilities. 119, 195

IEEP
An informal early experience program by invitation, rather than a formal or public early experience program (EEP). 94,

iff
if and only if.. 176

Implicit feedback
Implicit Feedback and automatically collected information about software usage [34].

In-app analytics
Usage-based analytics recorded by incorporating software into a mobile app. The analytics may be in the form of source code files, a library, or an SDK. Often they are provided by commercial specialist service providers. Some commercial providers provide their library or SDK as opensource code, packaged as a library or SDK, others provide a binary file.. 23, 47, 133–136, 177, 203, 239

Interaction screen
Interaction screen conveys the screen as both an output and input device (rather than the term touch screen) 

ISO
International Organization for Standardization (https://www.iso.org/home.html), 310

iTools
Improve the mobile analytics Tools and associated artefacts. One of the six perspectives identified in this research. 179

iUse
Improve the Use, the process, of mobile analytics tools and associated artefacts. One of the six perspectives identified in this research. 131

J
Jira
“Jira is a proprietary issue tracking product developed by Atlassian that allows bug tracking and agile project management.”, source: https://en.wikipedia.org/wiki/Jira_(software). 144, 146, 166, 213

JVM
Java Virtual Machine, for the purposed of this thesis the JVM runs Android apps written in Java and Kotlin. It is not used to run native code. 135, 145, 162

K
Kiwix
Kiwix refers to several related things: to the project github.com/kiwix/kiwix-android/, to the foundation www.kiwix.org/en/, and to the core Android app which is one of the case studies.

Kiwix custom app
The Kiwix Android codebase is used to create a general purpose app, called Kiwix, and a range of self-contained custom apps that include a single collection of material. Custom apps include those with medical articles, known as WikiMed, and other specialist areas. The apps share a common codebase and a consistent internal technical structure. 245

L
L1 theme
L1 theme: a lower-level theme that emerged from the findings. L1 themes are more detailed than L2 themes. 179

L2 theme
L2 theme: a higher-level theme that groups several L1 themes. 179

M
*M This term is originated by Prof. Arosha K. Bandara, thank you.
Mobile Analytics Policy
Definitions the ‘rules of engagement’ when incorporating mobile analytics as part of the development, maintenance and where appropriate the operation of a mobile app.

Mobile Analytics Services
Services provided using Mobile Analytics Tools. Typically mobile analytics services are provided by the owners of the intellectual property rights for the particular mobile analytics tool being used, for instance Sentry provide their mobile analytics service using software they own, they also make their software available as open source so others may be able to provide similar services subject to the licensing agreements of the various software components.

Mobile Analytics Strategy
Describes how the team intend to work within their mobile analytics policy to achieve the aims and objectives of using mobile analytics.

Mobile Analytics Tools
Software tools designed to collect information on one or more mobile platforms. They include Mobile App Analytics Tools and Mobile Platform Analytics Tools.

Mobile App Analytics Tools
“Mobile app analytics tools collect and report on in-app data pertaining to the operation of the mobile app and the behavior of users within the app, as well as aggregate market data on apps across public app stores” [360]. They are a subset of Mobile Analytics Tools.

Mobile Platform Analytics Tools
Are similar to Mobile App Analytics Tools, however, the data is collected by the platform, outside the app. Some of the data may be similar, some will only be available to either the platform or the app.

MSR
Mining Software Repositories is both a conference series and a community of researchers interested in that topic. www.msrconf.org/.

MTBF
Mean Time Between Failure.

N
Native Code
Written in C++ and native to the computer (mobile device) architecture.

npm
npm is used to manage and publish software packages. Contrary to popular belief, npm is not in fact an acronym for “Node Package Manager”; It is a recursive bacronymic abbreviation for “npm is not an acronym” (if the project was named “niniaa”, then it would be an acronym.) See: github.com/npm/cli#is-npm-an-acronym-for-node-package-manager.

O
OBB
Android Opaque Binary Blob File [361], used to package expansion files for Android APK files [362].

OEM
Original Equipment Manufacturer (OEM) manufacture devices, such as smartphone and tablet devices.

OKR
Objective and Key Result.

OOM
OutOfMemoryException Error, abbreviated to OOM. Oracle provides a guide oriented at programmers at docs.oracle.com/javase/8/docs/technote/guides/troubleshoot/memleaks002.html.

Operational Analytics
“Operational analytics: Provides visibility into the availability and performance of mobile apps in relation to device, network, server and other technology factors. Operational analytics are essential to capture and fix unexpected app behavior (such as crashes, bugs, errors and latency) that can lead to user frustration and abandonment of the app.” [363].
**OSX**

OSX is one of the many commonly-used terms that references the operating system Apple provides with Apple computers. Alternative terms include OS X, and macOS.

**PFOD**

Probability of Failure On Demand. 37, 187, 310

**PhET**

PhET is a project that creates and provides interactive simulations on computers for science and mathematics, free of charge. PhET stands for Physics Education Technology [364]. More information on the project is online at [https://phet.colorado.edu/](https://phet.colorado.edu/). The Kiwix project created the first mobile app for the project with the project’s permission. xvi, 205, 206, 239

**PII**

PII is “Personally Identifiable Information; Any representation of information that permits the identity of an individual to whom the information applies to be reasonably inferred by either direct or indirect means.” [365]. 132, 135, 136, 190

**Platform**

In the context of this research, a platform combines an ecosystem that encourages external developers, a mobile operating system, and various APIs. A useful overview of mobile platforms is ‘Mobile platforms and ecosystems’. 13,

**Platform analytics**

Usage-related analytics for various app-related information. The analytics are obtained by a mobile app platform, such as Google, on end-user devices (including smartphone and tablet devices), transmitted over an internet connection, and then processed by the platform provider to generate the analytics. Aspects of these analytics may be shared with various parties. . 47

**PLR**

Pre-launch report (PLR) This term is occasionally used in Google Android materials, for example in [367]. 7, 150, 153

**Pocket Code**

Pocket Code provides a visual programming environment where users can freely create, edit, execute, share, and remix programs with others. [play.google.com/store/apps/details?id=org.catrobat.catroid](play.google.com/store/apps/details?id=org.catrobat.catroid). It is one of the flagship mobile apps developed through the project. 134

**Pocket Paint**

Pocket Paint is a visual paint editor. It is also integrated into Pocket Code and can also be used on its own. [play.google.com/store/apps/details?id=org.catrobat.paintroid](play.google.com/store/apps/details?id=org.catrobat.paintroid). It is a popular mobile app developed through the project. 113

**Pre-launch report**

Pre-launch reports as provided by Google Play Console include various static analysis checks of the contents of the binary file of the app together with the results from pre-launch testing. 27, 192, 203, 205

**Pre-launch testing**

Pre-launch testing, automated monkey testing where a release candidate Android app is exercised interactively through the GUI of the app. The results include the application log, screenshots, and a video recording of the testing. It does not include the script so the tests cannot be directly reproduced. 7, 200, 250, 309

**Pull Request**

A Pull Request is a mechanism used by programmers to collect various source code and related files into a single request for other programmers to review. GitHub provides lots of information online about them [https://docs.github.com/en/pull-requests](https://docs.github.com/en/pull-requests). 244

**QoE**

Oft used in Mobile Telecommunications to measure network transmission characteristics. Here used so we can more easily identify and consider the quality of experiences such as User Experience. In the context of this research the focus is on perceived experience as perceived by end users of an app.
Quality-in-Use
Quality-in-Use in this thesis is based on the use of the term in the International Organization for Standardization (https://www.iso.org/home.html) (ISO) 9126/250xx Standards. In this research, Stability also pertains to Quality-in-Use models even though ISO 250xx doesn’t explicitly include them, possibly as the term stability was not a popular term in software quality, it has become more prevalent since the adoption of the term stability by HP and subsequently by Google in the context of measuring the quality of Android apps.

Reliability
A measure of software quality which compares the outcomes of software behaviour which can be reliable, or unreliable. The two main measures are MTBF and PFOD. One or other of these are used at any one time, not both. See Section 3.2.3. Stability and Reliability for a comparison with stability.

Risk
“A risk is an unwanted event that has negative consequences.” [368].

ROI
Return on Investment.

SAMOA
Software Analytics for MOBILE Applications, source samoa.inf.usi.ch/about/. 65, 66

SDK
Software Development Kit. 41, 42, 57, 60, 62, 64, 66, 93, 119, 121, 125, 126, 134–137, 139, 154, 158–163, 175–177, 180–186, 188, 190, 191, 201, 202, 209, 211, 214, 217, 221, 222, 224, 225, 228, 231–234, 236, 239

Service Provider
An organisation, often commercial, which includes software, and online services (and the people who provide these) that offers developers optional facilities such as in-app analytics, crash reporting, messaging, feedback mechanisms, etc.

Sharded
Something that is split into separate shards, for instance to improve capacity and/or scaling of work. Sharding is a term particularly used in database terminology in terms of how to segment data processing. 11

SLR
Systematic Literature Review. 34, 40

SNAFU
SNAFU describes a common, normal condition where things are generally messed up. The source comes from the military https://english.stackexchange.com/a/310644/433415 where I learned it. 131, 137, 138

Stability
A software quality identified initially by HP as part of their FunDex score [65]. The same term was subsequently used by Google to identify and measure two related indications of software failures: crashes and freezes [66]. These type of failure are both measures of reliability in terms of Probability of Failure On Demand (PFOD). See Section 3.2.3. Stability and Reliability for a comparison with reliability.

Statistics-based debugging
Harnesses automated data collection at a global scale to help programmers more effectively improve system quality [69], and more details are available in [70].

SUT
System under test. 81,

Technology-facing
The term stems from Brian Marick and his blog posts on Agile software development including software testing. An early example from his old blog is: “A technology-facing test is one you describe with words drawn
from the domain of the programmers” [www.exampler.com/old-blog/2003/08/21.1.html]. The concepts are well explained in Chapter 6 of [252] on pp. 97-98 and beyond of that book. 8, 175, 176

U
uArtefacts
Understand development Artefacts, including the mobile app, pertaining to the use of mobile analytics. One of the six perspectives identified in this research. 157

UI
User Interface. 9

uTools
Understand the mobile analytics Tools and associated artefacts. One of the six perspectives identified in this research. 179

uUse
Understand the Use, the process, of mobile analytics tools and associated artefacts. One of the six perspectives identified in this research. 131

UX
User Experience. 132, 236

V
Vectored questioning
A qualitative method to focus questions asked to people on projects with a focus on answering the research questions.

Vitals Scraper
Opensource software developed as part of this research to facilitate the downloading, analysis, and preservation of various reports in Google Play Console and Android Vitals. The source code project is available online [https://github.com/commercetest/vitals-scraper]. 243

W
WebView
Android WebView is a pre-installed system component from Google that allows Android apps to display web content. (Source play.google.com/store/apps/details?id=com.google.android.webview.) xix, 139, 148, 166, 174

WER
Windows Error Reporting - a service developed by Microsoft used to find bugs at scale in the Windows Operating System. More information on how they do so available in [69, 70]. 41
Alphabetical Index

IllegalStateException, 144
IndexOutOfBoundsException, 144, 205
NullPointerException, 144, 192, 202
UnsatisfiedLinkError, 245
getHistoricalProcessExitReasons, 168
IndexOutOfBoundsException, 215
NullPointerException, 205
OutOfMemoryError, 149
Action research, 78, 134
Amazon, 224, 225
Fire OS, 3
Amplitude, 124, 163, 234
Analysis
Sourcecode, 120
Analytics
Firebase, 122
Android Studio, 213
Android Test Automation
Monkey, 50, 54, 63, 148, 250
Robo., 63
AndroidCrashDummy, 96
ANR watchdog, 183
App store, 16
App Center, 93
Microsoft, 125
App Pulse Mobile, 64
App store, 14
AppAnnie, 61
AppBrain, 7, 197, 199
AppClix, 65
Apple
iOS, 3, 6, 61, 64
AppPulse Mobile, 127, 182, 232
AppSee, 127
Appsee, 200
Apptentive, 60
Azetone, 127, 207, 231
Bad behavior threshold, 153
BBC
iPlayer app, 4
Beacon-finding, 78
Bumble, 168
Caiipa, 58
Case studies, 8
Chrome.apk, 148
Countly, 126, 127, 207, 225, 231
Covid-19, 3, 74, 114
Crash, 5, 8
Analytics, 6
Crash reporting, 135
Crashlytics, 57, 121, 181, 192, 201, 203, 211, 271
Fabric, 93, 135
Firebase, 135
CrashProbe, 214
CrashProbe opensource project, 125
CrashScope, 52, 236
Desktop app ecosystem, 33
Development artefacts, 73
DevOps, 26
EduVPN, 127
eduVPN, 252
Engineering leadership, 143
Error
Analytics, 6
Exodus Privacy Project, 86
Expo framework, 174
F-Droid, 66
Fabric, 185
Fabric Crashlytics, 116, 137, 138, 166, 181, 185, 191, 211, 278
Facebook, 5, 202
Failure cluster, 80, 243
Feedback mechanisms, 17, 24, 78, 84, 237
Field experiment, 90, 118, 119
FindBugs, 60
Firebase, 122, 152, 161, 163, 185, 191, 200, 225
Firebase Crashlytics, 116, 154, 160, 175, 185, 186, 194, 203, 213
Firebase Test Lab, 203
Firebox, 72

Flaw
- Gaps in the data, 208, 260
- Inconsistent data ranges, 208, 257
- Incorrect last update date, 208
- Incorrect ranges, 208
- Lack of reports (despite usage), 208
- Mismatches between analytics tools, 208
- Missing URL parameters, 208
- Negative populations, 208, 257
- No status reporting, 208, 260
- Poor grouping of clusters, 208
- Repeated crash clusters, 208
- Repeated graphs, 208, 257
- Second rank usurps first, 208, 257
- Stale data, 208, 257
- System failures, 208, 260
- Testing discouraged, 208
- Unexplained decline in userbase, 208

Flurry, 160
Flurry Analytics, 61, 65

getHistoricalProcessExitReasons, 183
GitHub, 140, 144
GitHub Projects
  - Android Crash Dummy, 161
  - Android Log Assert, 161
GitLab, 144
Google Analytics, 30, 64, 136, 191, 196, 203
Google Chrome, 72
Google Play, 4–7, 13, 14, 54, 106, 172, 173, 192, 197, 223, 230
Release Management, 192, 195, 210
Google Play Store, 199
Gradle, 176
Grey data, 71, 73, 81, 84
Grey literature, 71, 73, 81, 84
GTAF, 92, 100, 133–135, 138, 142–144, 146, 149, 150, 183, 184, 204, 213, 222, 226, 247, 252
Hackathon, v, 91, 112, 113, 133, 145, 154, 166
HarmonyOS, 225
Heatmaps, 58, 62
HockeyApp, 57, 125
Huawei, 224
  - HarmonyOS, 3
Informed consent, 169, 230, 231

Insight (Analytics), 64, 65
iOS, 225
Ishikawa (fishbone) diagram, 90
Iteratively, 94, 119, 124, 189, 190, 209, 214, 217, 234
Iteratively’s, 223
Jenkins, 116
Kotlin, 135, 162, 163, 175, 182
LeakCanary, 150
Local app experiments, 77, 81, 257
logcat, 53
Logging, 26, 186

Meta-data, 185, 186, 191, 217, 232
Method
  - Across case comparisons, 77, 82, 197, 245, 255, 260
  - App binary analysis, 86, 87
  - Ask the app devs, 84, 245, 247, 257
  - Ask the tool devs, 85, 255, 260
  - Beacon-finding, 140, 247, 255
  - Drill-down, 140, 245, 248, 255, 258
  - Grey data, 85, 247
  - Grey literature, 85, 247
  - Grey material, 86
  - Sense-building, 77, 255
  - Sense-making, 36, 77, 78, 82, 141, 255
  - Source code analysis, 86, 245
Micro experiment
  - Android Crash Dummy, 119, 275
  - Contribution to opensource mobile analytics, 278
  - IDoT, 119, 124, 195
  - Travel Europe, 119, 211
  - Zipternet, 119, 209, 276
Micro experiments, 81, 119, 231, 271
Microsoft App Center, 118, 134, 140, 146, 160, 192, 196, 201, 207, 211, 214, 231, 232, 273
MixPanel, 127, 191
Mixpanel, 134
Mobile analytics, 9
Monkey Testing, 24, 50, 52, 54, 56, 63
MonkeyLab, 52
MoodSpace, 202