
INSIGHTS INTO BUILDING APPS WITH MILLIONS OF USERS
BUILDING MOBILE APPS AT SCALE
39 Engineering Challenges
SAMPLE
INDUSTRY PRACTICES USED BY LARGE MOBILE TEAMS
GERGELY OROSZ

Building Mobile Apps at Scale (Sample)

39 Engineering Challenges

ISBN: 978-1-63795-844-5 (ebook)

FIRST EDITION, v1.0

www.MobileAtScale.com

Table of Contents

Introduction

I have noticed that while there is a lot of appreciation for backend and distributed systems challenges, there is often less empathy for why mobile development is hard when done at scale. Building a backend system that serves millions of customers means building highly available and scalable systems and operating these reliably. But what about the mobile clients for the same systems?

Most engineers who have not built mobile apps assume the mobile app is a simple facade that requires less engineering effort to build and operate. Having built both types of systems, I can say this is not the case. There is plenty of depth in building large, native, mobile applications, but often little curiosity from people not in this space. Product managers, business stakeholders, and even non-native mobile engineers rarely understand why it "takes so long" to ship something on mobile.

This book collects challenges engineers face when building iOS and Android apps at scale. By scale, I mean having numbers of users in the millions and being built by large engineering teams. These teams launch features continuously and still ensure the app works reliably, and in a performant way.

This book is a summary of the current industry practices used by large, native mobile teams and points to some of the common approaches to tackle them. Much of the experience conveyed in this book comes from my time working at Uber on a complex and widely-used app. More than 30 other engineers working in similarly complex environments have contributed their insights; engineers building apps at the likes of Twitter, Amazon, Flipkart, Square, Capital One and many other companies.

I hope this book helps non-mobile engineers build empathy for the type of challenges and tradeoffs mobile engineers face and be a conversation starter between backend, web, and mobile teams.

Acknowledgements

The book has been written with the significant input and reviews of more than 30 mobile engineers and managers, many of them deep experts in their respective fields. Thank you very much to all of them. If you are on Twitter, I suggest you follow them:

- Abhijith Krishnappa (Halodoc)
- Andrea Antonioni (Just Eat)
- Andreea Sicaru (Uber)
- Andy Boedo (RevenueCat, Elevate Labs)
- Ankush Gupta (Quizlet)
- Artem Chubaryan (Square)
- Barnabas Gema (Shapr3D)
- Barisere Jonathan (Sprinthubmobile)
- Corentin Kerisit (Zenly)
- Dan Kesack (DraftKings)
- Edu Caselles (Author: The Mobile Interview, Funding Circle)
- Emmanuel Goossaert (Booking.com)
- Franz Busch (Sixt)
- Guillermo Orellana (Monzo, Skyscanner, Badoo)
- Injy Zarif (Convoy, Microsoft)
- Jake Lee (Photobox)
- Jared Sheehan (Capital One)
- Javi Pulido (Plain Concepts)
- Jorge Coca (VG Ventures)
- Julian Harty (previously Google, eBay, Badoo, Salesforce, Klarna, ServiceNow, and others)
- Leland Takamine (perf.dev, Uber)
- Matija Grcic (EMG Consulting)
- Michael Bailey (GDE, American Express)
- Michael Sena (Tesla, Amazon)
- Nacho Lopez (Twitter, Facebook, Yahoo)
- Patrick Zearfoss (Capital One)
- Praveen Sanap (Carousell)
- Paul Razgaitis (Cameo, Braintree, Venmo)
- Robin van Dijke (Uber, Apple)
- Rui Peres (Sphere, Babylon Health)
- Sabyasachi Ruj (Flipkart, CloudMagic, Webyog)
- Sathvik Parekodi
- Tuğkan Kibar
- Tuomas Artman (Linear, Uber)
- Wouter van den Broek

Thank you to Emmanuel Goossaert for writing most of Chapter 10: Third-Party Libraries and SDKs.

Special thanks to the editor of the book, Dominic Gover at Best English Copy for helping create a book that is pleasant to read.

About the Author

Gergely has been building native mobile apps since 2010, starting on Windows Phone, later on iOS, and Android. Starting from one-person apps, he worked with small teams at Skyscanner, to hundreds of engineers working on the same codebase at Uber.

At Uber, he worked on the Rider and Driver app rewrites, both projects involving hundreds of mobile engineers. The apps he worked on had more than 100 million monthly users in more than 60 countries, with several features built for a single country or region.

You can read books he has written, browse The Pragmatic Engineer Blog he writes and connect with him on social media channels.

When Things are Simple

Let's address the elephant in the room: the frequent assumption that client-side development is simple. The assumption that the biggest complexity lies in making sure things look good on various mobile devices.

When the problem you are solving is simple, and the scope is small, it is easier to come up with simple solutions. When you are building an app with limited functionality with a small team and very few users, your mobile app should not be complicated. Your backend will likely be easy to understand. Your website will be a no-brainer. You can use existing libraries, templates, and all sorts of shortcuts to put working solutions in place.

Once you grow in size — customers, engineers, codebase and features — everything becomes more complex, more bloated, and harder to understand and modify, including the mobile codebase. This is the part we focus on in this book; when things become complex. When your app gets big, there are no silver bullets that will magically solve all of your pain points, only tough tradeoffs to make.

This book begins at the point at which your app stops being simple.

PART 1: Challenges Due to the Nature of Mobile Applications

People who have not done mobile development often assume that most challenges of native apps are similar to those on the web.

This could not be further from reality.

Mobile engineering has close to a dozen unique challenges that exist neither on the web, nor on the backend. Most of these relate to the binary distribution of mobile apps; the need to work in low connectivity setups, and to incorporate unique capabilities like push notifications, deeplinks or in-app purchases.

In this part, we will go through the challenges that are new to people who have not worked in the mobile domain, both for software engineers, but also for engineering managers, product managers and business stakeholders.

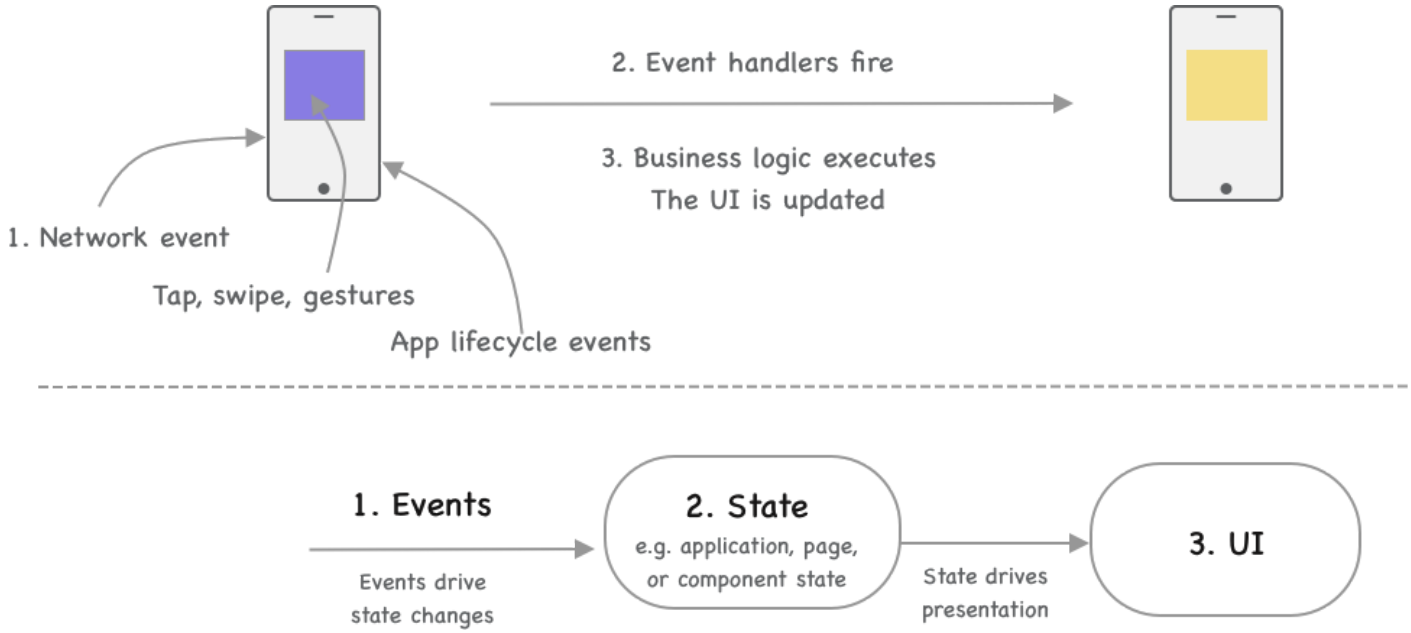
This sample PDF contains 3 out of the 12 chapters of this part in the book.

Chapters not included in this sample:

- *4. Deeplinks*
- *5. Push and Background Notifications*
- *6. App Crashes*
- *7. Offline Support*
- *8. Accessibility*
- *9. CI/CD & The Build Train*
- *10. Third-Party Libraries and SDKs*
- *11. Device and OS Fragmentation*
- *12. In-App Purchases*

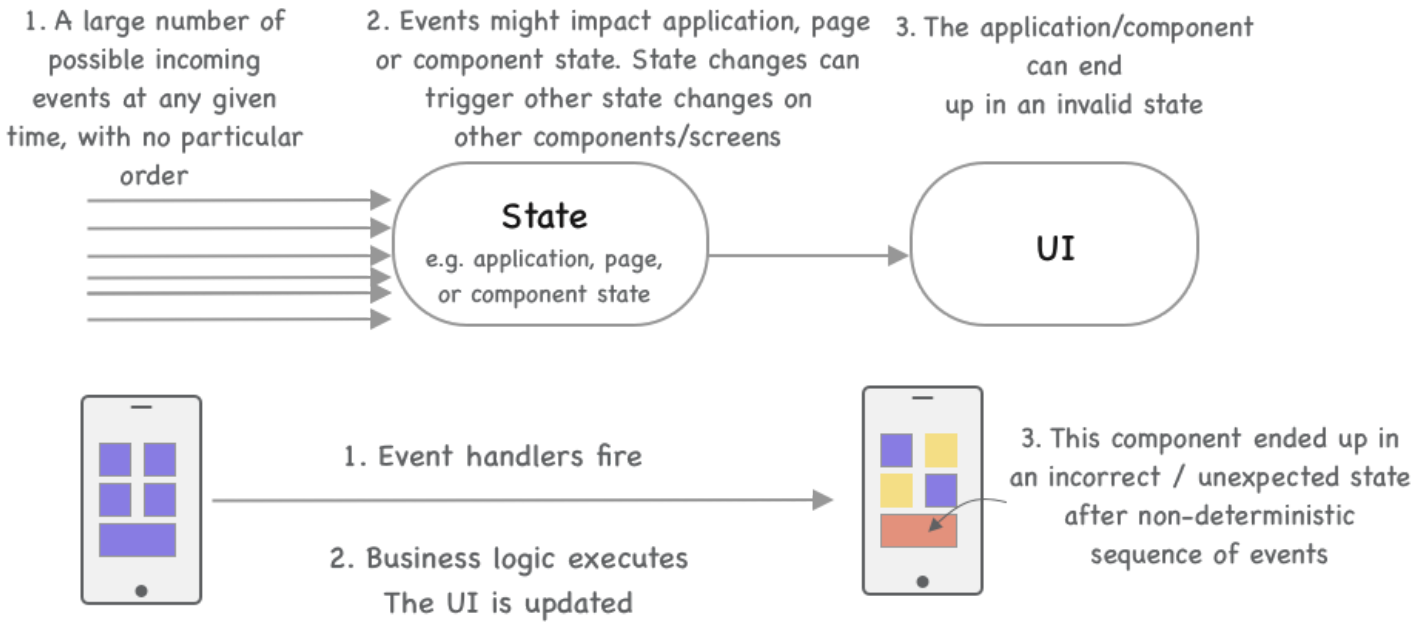
1. State Management

State management is the root of most headaches for native mobile development, similar to modern web and backend development. The difference with mobile apps is how app lifecycle events and transitions are not a cause for concern in the web and backend world. Examples of the app-level lifecycle transitions are the app pausing and going to the background, then returning to the foreground or being suspended. The states are similar, but not identical for iOS and Android.



Events driving state changes in mobile apps

Events drive state changes in most mobile apps. These events trigger in asynchronous ways such as application state changes, network requests or user input. Most bugs and unexpected crashes are usually caused by an unexpected or untested combination of events and the application's state becoming corrupted. State becoming corrupted is a common problem area with apps where global or local states are manipulated by multiple components unknown to each other. Teams that run into this issue start to isolate component and application state as much as possible and tend to start using reactive state management sooner or later.



A common root reason for exotic bugs in complex apps: non-deterministic events put parts of the app in invalid states

Reactive programming is a preferred method for dealing with a large and stateful app, in order to isolate state changes. You keep state as immutable as possible, storing models as immutable objects that emit state changes. This is the practice used at Uber, is the approach Airbnb takes, and is how N26 built their app. Though the approach can be tedious in propagating state changes down a tree of components, the same tediousness makes it difficult to make unintended state changes in unrelated components.

Applications sharing the same resources with all other apps, and the OS killing apps at short notice, are two of the biggest differences between developing for mobile versus developing for other platforms, like backend and the web. The OS monitors CPU, memory, and energy consumption. If the OS determines that your app is taking up too many resources in the foreground or the background, then it can be killed with little warning. It is the app developer's responsibility to react to application state changes, save state, and to restore the app to where it was running. On iOS, this means handling app states and transitions between them. On Android, you need to react to changes in the Activity lifecycle.

Global application state, such as permissions, Bluetooth and connectivity state, and others, brings an interesting set of challenges. Whenever one of these global states changes, for example, the network connectivity drops, then different parts of the app might need to react differently.

With global state, the challenge becomes deciding which component owns listening to these state changes. At one end of the spectrum, application screens or components could listen to global state changes they care about; resulting in lots of code duplication, but components handling all of the global state concerns. At the other end, a component could listen to certain global state changes and forward these on to specific parts of the application. This might result in less complex code, but now there is a tight coupling between the global state handler and the components it knows.

App launch points like deeplinks or internal shortcut navigation points within the app also add complexity to state management. With deeplinks, the application state might need to be set up after the deeplink is activated. We will go into more detail in the Deeplinks chapter.

2. Mistakes Are Hard to Revert

Mobile apps are distributed as binaries. Once a user updates to a version with a client-side bug, they are stuck with that bug until a new version is released and the user updates.

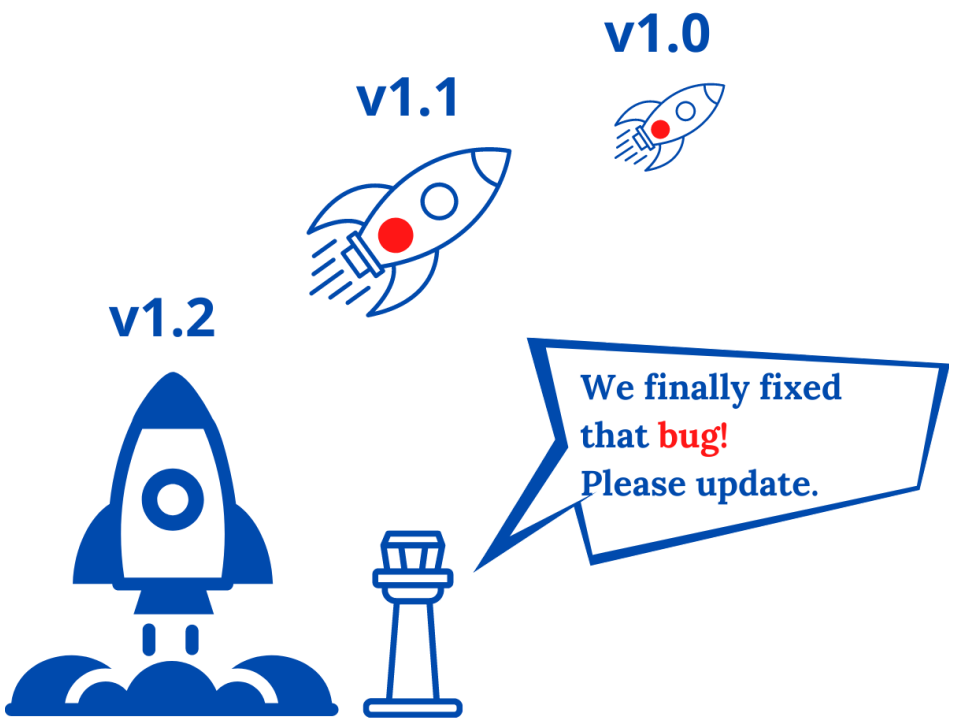
Multiple challenges come with this approach:

- **Both Apple and Google are strict on allowing executable code to be sent to apps.** Apple does not allow executing code that changes functionality in their store guidelines and Google can flag unrelated executable code as malware, as per their program policy. This means you cannot just remotely update apps. However, pushing bug fixes that revert broken functionality should be within both stores' policies: for example, when using feature flags. At the same time, Apple does allow executing non-native code like JavaScript, which is why solutions like Codepush are gaining popularity. Codepush allows React Native or Cordova apps to deliver updates on the fly. At Uber, we built a homegrown solution along the same lines, as several other companies have done.
- **It takes from hours to days to release** a new app version on the store. For iOS, manual reviews happen for all apps, taking 24-48 hours to complete. Historically, every review had the possibility of rejection. As of June 2020, Apple has changed guidelines, so bug fixes are no longer delayed over guideline violations, except for legal issues. On Android, manual reviews do not always happen, but when they do, they can take more than seven days.
- **Users take days to update to the latest version** after a new version is published to the app store. This lag is true even for users with automated updates turned on.
- **You can not assume that all users will get this updated version, ever**. Some users might have automated updates disabled. Even when they update, they might skip several versions.

Chuck Rossi, part of release engineering at Facebook, summarizes what it is like to release for mobile on a Software Engineering Daily podcast episode:

> *"It was the most terrifying thing to take 10,000 diffs, package it into effectively a bullet, fire that bullet at the horizon and that bullet, once it leaves the barrel, it's gone. I cannot get it back, and it flies flat and true with no friction and no gravity till the heat death of the universe. It's gone. I can't fix it."*

This means that all previous versions of your app need to be supported indefinitely and in theory, you should do this. The only exception is if you put homegrown controls in place and build a force update mechanism to limit the past versions to support. Android supports in-app updates in the Play Core library. iOS does not have similar native support. We will cover more on this in the Forced Upgrading chapter.



Fixing a bug in a mobile app

Assuming you have an app with millions of users, what steps can you take to minimize bugs or regressions from occurring in old versions?

- **Do thorough testing** at all levels. Automated testing, manual testing, and consider beta testing with easy feedback loops. A common approach at many companies is to release the beta app to company employees and beta users for it to "bake" for a week, collecting feedback on any issues.
- **Have a feature flagging system** in place, so you can revert bugs on the fly. Still, feature flags add further pain points. We will discuss these points in the Feature Flag Hell chapter.
- **Consider gradual rollouts**, with monitoring to ensure things work as expected. We will cover this topic in the Analytics, Monitoring and Alerting chapter.
- **Force upgrading** is a robust solution, but you will need to put one in place, and some customers might churn as a result. We will go deeper on this in the Forced Upgrading chapter.

3. The Long Tail of Old App Versions

Old versions of the app will stay around for a long time, up to several years. This time frame is only shorter if you are one of the few teams that put strict force app upgrade policies in place. Apps that have a rolling window of force upgrades include Whatsapp and Messenger. Several others use force upgrades frequently, like banking apps Monzo or American Express.

While most users will update to new app versions in a matter of days, there will be a long tail of users who are several versions behind. Some users disable automatic updates on purpose, but many who do not update are blocked because of old phones or OSes. At the same time, old app versions are unlikely to be regularly tested by the mobile team because it is a lot of effort, with little payoff.

Even a non-breaking backend change can break an older version of the app - such as changing the content of a specific response. A few practices you can do to avoid this breakage:

- **Build sturdy network response handling and parsing, using dedicated tooling** that solves these problems. I prefer strongly typed, generated contracts between client and backend like Thrift, GraphQL, or other solutions with code generation, over REST interfaces that you need to validate manually, which is bound to break when someone forgets to update the parsing logic on mobile.
- **Plan well in advance for breaking backend changes.** Have an open communications channel with the backend team. Have a way to test old app versions. Consider building new endpoints and not retiring old ones until a forced upgrade moves all current app users off the old endpoint.
- **Version your backend endpoints** and create new versions to accommodate breaking changes. When making breaking changes, you will usually create a new endpoint and mark the existing one as deprecated. Note that in case of using GraphQL, this might not apply, as GraphQL takes a strong stance against versioning.
- **Proceed with caution when deprecating endpoints** on the backend. Monitor the traffic, and have a migration plan on how to channel requests, if needed.
- **Track usage stats on an app version level.** What percentage of users is lagging three or more versions behind? Once you have this data, it is easier to decide how much effort to dedicate towards ensuring the experience works well on older versions.
- **Put client-side monitoring and alerting in place.** These alerts might be channeled to a dedicated mobile on-call, or just the normal on-call. We will dive more into this in the Analytics, Monitoring and Alerting chapter.
- **Consider doing upgrade testing**, at least for major updates. Upgrade testing is expensive, hard to automate, and there might be several permutations to try. Teams rarely do it because of this overhead.

PART 2: Challenges Due to App Complexity

Things start to get interesting as an app grows. You keep adding new features to the app, while also tweaking the existing ones. Soon, the app that used to be a few screens gets so complex that, if you were to print out screens on a navigation flow chart, it would cover the whole wall.

When working with a large and complex app, you do run into additional challenges. How do you deal with increasingly complicated navigation patterns? What about non-deterministic event combinations? How do you localize across several languages, and how do you scale your automated and manual tests?

Let's jump in.

This sample PDF contains 1 out of the 6 chapters for this part in the book.

Chapters not included in this sample:

- *13. Navigation Architecture Within Large Apps*
- *14. Application State & Event-Driven Changes*
- *16. Modular Architecture & Dependency Injection*
- *17. Automated Testing*
- *18. Manual Testing*

15. Localization

Both iOS and Android offer opinionated ways to implement localization. iOS supports exporting localizations for translation, while Android builds on top of resource strings. The tooling is slightly different, but the concept is similar. To localize your app and define the strings, you want to localize and ship the localized strings as a separate resource in the binary. Still, with large apps and many locales, you quickly run into challenges with this workflow.

Deciding what to localize in the app versus doing so on the backend is one of the first challenges any growing app will face. By using the “default” iOS and Android localization approach, resources you localize within the app will be "stuck" with the binary: you cannot change phrases or update mistakes. If you implement serving localized strings from a vendor solution, or your backend, you have more flexibility in this regard and you only need to do one localization pass for both iOS and Android.

The topic of how "smart" the mobile app should be and what strings should live here, versus just getting them from the backend, runs deeper. We will go into more detail in Part 3: Backend-driven mobile apps. The more localization the backend does, the better. Backend-heavy localization keeps client-side logic low and reduces the number of resources for localization on mobile. While delegating more localization to the backend usually needs work, both on the iOS and Android app workflows, it leads to easier maintainability in the long run.

When supporting a large number of locales, you need to ensure that all localization translations are complete before shipping to the app store. You might be using third-party localization services or an in-house team to do so. Assuming you have a build train, you want to allow the train to proceed only after all new resources have been localized.

Ensuring iOS and Android use the same language and localization is yet another challenge. Especially for larger brands, it is important that both the iOS and Android apps use the same — or at least similar — languages. A consistent language is not only beneficial for the brand, it also helps customer support handle issues reported by users. It is hard to ensure this consistency without some shared localization tooling or the iOS and Android teams working closely with each other. Having the same designer and PM overseeing both apps also helps. Using the same localization IDs or keys is a good way to reduce duplication and this needs to be done through the iOS and Android teams agreeing on conventions.

At Uber, we used an in-house localization tool across iOS, Android, backend, and web. This tool was integrated to generate iOS and Android resources and to track the completeness of localization. Was it overkill to build a dedicated tool just for localization? For Uber’s sophisticated use cases years back, I do not think so. Back then, there were few to no good localization tools across all stacks.

The space has changed since, though, and there are many localization vendors in the mobile space. Before building your own, it is worth seeing if you can find one that fits your needs.

Localized custom fonts is a frequently overlooked area. It is common for larger companies and those with strong brand identities to use custom fonts. However, this custom font does not always support every glyph for an app's supported set of languages.

It is tricky enough to confirm which languages and characters your custom font is missing support for. This is even more of an issue when using these fonts to display user-generated content, as you do not have control over this input. Once you have figured out your use cases and edge cases, you either need to have this support added or — more frequently, — use a different font for unsupported locales. It gets even more difficult to manage a consistent, cross-platform mapping of fonts and languages, based on which languages each font supports.

When using custom fonts, you need to include these in the binary, adding to the app's size. We will discuss app size in more detail in Part 3 of the series.

Currencies formatted differently on iOS and Android on certain locales is another pain point that multi-language, multi-platform apps displaying monetary values encounter. Web has a similar problem with the inconsistency; Indian rupee values are a good example of this. The problem is especially pronounced when using currency types to add or subtract values. A possible workaround is for the backend to format all currency data to the locale, and the client not doing any math operations, or formatting currency strings.

Date and time formatting will have similar challenges to currencies. Different countries and regions will display dates and times in various ways. You need to make sure the app accounts for this.

Supporting right-to-left (RTL) languages can often go beyond just translations. Most people reading this book will either be used to LTR and Latin-based languages or speak it fluently enough. However, when designing UIs for RTL languages like Arabic, Hebrew, and others, it is not just strings that need to be mirrored; the layout of the app might need to be changed. "Mirroring" the layout is a common approach, but it might result in strange UIs for native speakers and users. It is best to have locals involved in this process, both for development and for testing.

Unique locales are always worth additional attention. In my experience, Japanese and German are locales are worthy of even more checks because they can both be more verbose than English, and you want to make sure the layouts, paddings, and line breaks still work for these locales.

The Skyscanner app with English, Japanese and German languages. German is often a good choice to stress test the app, as the language is more verbose.

Testing localization is no small effort. In an ideal world, a native speaker would go through every flow of your app after each localization change. In reality, this rarely happens because it is too expensive to do. Most large companies rely on beta testers using the app with different locales to report glaring inconsistencies. For example, at Uber, we could rely on this method to spot localization issues early. Thousands of Uber employees dogfooded the app every week, alongside beta testers. Localization issues almost always got caught before the app got pushed to the app store.

You might need to define a workflow to validate localization changes. What should happen when a localized string changes in the app? Should this trigger an action to manually inspect the change? Should developers for the screen be notified? Should the release manager ship a new version of the app, even if there are no other changes? These might seem like minor questions. They are not. They are especially not when people translating strings work independently to engineers writing the code. Someone needs to define a workflow of not just adding, but updating and testing localization.

Snapshot testing is an underrated testing tool for localization. With snapshot tests, you can quickly and easily generate snapshots for screens in any locale, or even with pseudo-localization. Engineers can then spot layout issues much faster, and have reference images showcasing the issues. On top of helping engineers, you can share the snapshot test screenshots with people doing the translation, so they get additional context on how the translated text will appear.

Pseudo-localization is a smart way to test localization working, without going through the localization exercise. It means replacing all localizable elements with a pseudo-language that is readable by developers, but which contains most of the "tricky" elements of other languages, such as special characters or longer strings.

For example, a pseudo-localized version of "Find Help" could appear as "[ƒîîîñ̃ð̃ð̃ð̃ð̃ Ĥéééé̜ļþ]. Microsoft used this approach while developing Windows Vista and Netflix uses this approach for their product development cycle. Shopify built a ruby tool to generate strings like this, and you can find an npm package that was inspired by the approach at Netflix.

Phrases that should not be localized are one final edge case. At Uber, we decided not to localize certain brand terms like Uber Cash or Uber Wallet. Some teams were not aware of this request and went ahead and translated the strings in certain screens, meaning the owning teams had to test on all locales to find these issues. Several testers also reported the lack of these translations on a regular basis. This was a bit of noise we had to manage.

There are plenty of vendors who can help with mobile localization. Localization is rarely done in isolation only on iOS and Android; it is more commonly done as a whole, including the web, emails and other customer-facing properties. Localization vendors include POEditor, Loco, Transifex, Crowdin, Phrase, Lokalise, OneSky, Wordbee, Text United, and several others.

Further reading:

- Pseudo-localization from Netflix
- Why you should care about pseudo-localization from Shopify
- Design for internationalization from Dropbox
- Practical internationalization tips from Shopify
- Localizing across multiple platforms from Slack
- Localizing native apps with Localicious from Picnic
- Android localization guide from ProAndroidDev
- Using adaptive width strings for iOS localization from Daniel Martín

PART 3: Challenges Due to Large Engineering Teams

Building an app with a small team is very different than doing so with 10, 20, or more, mobile engineers. Lots of problems that seemed insignificant with a small team, become much larger.

Ensuring consistency in architecture becomes much more challenging. If your company builds multiple apps, how do you balance not rewriting everything from scratch while moving at a fast pace, over waiting on "centralized" teams?

The larger the mobile team, the more builds become a problem. Larger teams usually mean more code and slower build times. With a large team, the cost of a slow build can mean engineering months — or years! — wasted doing nothing. Finding tooling that supports a large mobile engineering team becomes increasingly difficult and some teams will resort to building custom solutions, when they cannot find vendors who support their scale of use cases.

Let's go through the challenges that become more pressing as your mobile engineering team grows in size.

This sample does not have any chapters from this part. The full book contains these chapters:

- *19. Planning and Decision Making*
- *20. Architecting Ways to Avoid Stepping on Each Other's Toes*
- *21. Shared Architecture Across Several Apps*
- *22. Tooling Maturity for Large Engineering Teams*
- *23. Scaling Build & Merge Times*
- *24. Mobile Platform Libraries and Teams*

PART 4: Languages and Cross-Platform Approaches

The tooling to build mobile apps keeps changing. New languages, frameworks, and approaches that all promise to address the pain points of mobile engineering keep appearing.

But which approach should you choose? Should you use Objective C or Swift to build iOS apps? Java or Kotlin for Android? Or should you invest in a cross-platform approach that promises a “write once, run everywhere” approach like Flutter, React Native, Cordova? Or perhaps settle for a middle ground, or reuse business logic written in Kotlin, C#, C++ or other languages?

In this section, we cover the most common industry choices regarding languages, frameworks, and cross-platform approaches when building mobile apps. There is no shortage of tools on how to build apps. It will be down to the constraints and tradeoffs you are willing to make in the approach you choose, for building across iOS, Android - and possibly other - platforms.

This sample does not have any chapters from this part. The full book contains these chapters:

- *25. Adopting New Languages and Frameworks*
- *26. Kotlin Multiplatform and KMM*
- *27. Cross-Platform Feature Development*
- *28. Cross-Platform App Development versus Native*
- *29. Web, PWA & Backend-Driven Mobile Apps*

PART 5: Challenges Due to Stepping Up Your Game

Your approach to mobile engineering changes when you are aiming to build not just a good enough mobile app, but a best-in-class experience. This change in approach might come as a result of your app serving millions of customers, or it can be because you want to make world-class mobile experiences part of your app's DNA from day one.

This part covers problems that "world-class" apps tackle from the early days. We will cover non-functional aspects like code quality, compliance, privacy, compliance. We will also go through experimentation and feature flag approaches that are table-stakes for innovative apps, and other areas you need to pay attention to, like performance, app size, or forced upgrading approaches.

Let's get started!

This sample PDF contains 2 out of the 10 chapters of this part in the book.

Chapters not included in this sample:

- *31. Feature Flag Hell*
- *32. Performance*
- *33. Analytics, Monitoring and Alerting*
- *34. Mobile On-Call*
- *36. Compliance, Privacy and Security*
- *37. Client-Side Data Migrations*
- *38. Forced Upgrading*
- *39. App Size*

30. Experimentation

Any company that has a mobile app that drives reasonable revenue will A/B tests even small changes. This approach allows for both measuring the impact for changes and ensuring there are no major regressions that impact customers — and revenue — negatively.

Feature flags are just the first necessary tool for an experimentation system. Controlled rollout via staging and user bucketing, analyzing the results, detecting and responding to regressions and post-experiment analysis all make up an advanced and powerful experimentation system.

When you are small, experimentation is easy enough, mostly because you rarely have more than a few experiments taking place. Compare this to an Uber-scale app, where there might be more than 1,000 experiments running at any given time, each experiment targeting different cities and target groups, and some experiments impacting each other.

Tooling is one part of the question. There are plenty of pretty mature experimentation systems out there — some built for native mobile — from the ground up.

In-house experimentation systems are common for larger companies for a few reasons:

- **Novel systems**. Many of these systems are novel, evolving as data science and engineering evolves within the company. There is often nothing as advanced on the market as the data science team wants.
- **Data sources** can come from many places, several of which are in-house. For example, you can directly link an experiment with revenue generated, and compare that to the treatment group's revenue.
- **Supporting many teams** in an efficient way is not something most third-party experimentation platforms do well.
- **Data ownership** is clear: all experiment data stays in-house.
- **Core capabilities** for tech companies are rarely "outsourced". As of today, the ability to rapidly experiment and make decisions based on data is large enough of an advantage to want to keep it in-house. Even if it means spending more money, an in-house solution can allow the company to stay ahead of the competition.

Even if there was an alternative solution that a company could buy, it would mean an expensive migration, and some in-house features might work differently, or not even exist. For example, Uber had unique regulatory requirements for certain cities and regions that had to be baked into how experiments were or were not rolled out to a certain region. This regulatory requirement was specific to the gig economy and to specific cities. It is highly unlikely that any experimentation platform on the market would know the context to be able to support that use case.

Companies which chose to keep experimentation in-house include Uber, Amazon, Google, Netflix, Twitter, Airbnb, Facebook, Doordash, LinkedIn, Dropbox, Spotify, Adobe, Oracle, Pinterest, Skyscanner, Prezi, and many others. Skyscanner is an example of a company who started with a vendor — Optimizely — before deciding to move experimentation in-house.

Motivations for moving to a vendor from in-house solutions are often cost-based (it is more expensive to operate and maintain the current system than using a vendor is), and standardization (having a unified platform instead of several teams building and maintaining custom tooling, and doing experiments in silos). GoDaddy is an example where they moved part of their experimentation to a vendor solution, in an effort to standardize across orgs, while keeping the feature flag implementation in-house.

Off-the-shelf experimentation and feature flag systems are plenty, and small to middle-sized companies and teams typically choose one of them. This is the point where building, operating and maintaining an in-house system could be more expensive and come with fewer features, than does choosing a vendor.

Popular vendor choices include Firebase Remote Config, LauchDarkly, Optimizely, Split.io and others. On top of a feature flagging system, many companies use a product analytics framework for more advanced analysis, Amplitude being a commonly quoted one.

The other difficulty is process; keeping track of experiments, and making sure they do not impact each other. For a small team, this is not a big deal. But when you have more than a dozen teams experimenting, you will encounter experiments impacting each other.

Process and tooling can both help with keeping track of experiments. Broadcasting upcoming experiments, data scientists or product managers syncing and a tool that makes it easy to locate and monitor active experiments can all help here. Most of these will be things you need to put in place as custom within the company, though.

Experimenting with every change is a common approach at companies with large apps, where the business impact of a change gone wrong could be bad. Uber is a good example where every mobile change needed to be reversible, and behind a feature flag. Even in the case of bug fixes, the fix would be put behind a feature flag, and rolled out as an A/B test. We would monitor key business metrics such as signup completion, trip taking rate or payments success rate, and keep track of any regression. All changes in the app needed to not reduce the key business metrics.

Was this approach an overkill? For a small application, it would be. However, at Uber, this approach helped us catch problems that resulted in people taking fewer trips. Even a 1% decline in trips would be measured in the hundreds of millions dollars per year, so there was reason to pay attention to this.

I recall shipping a bug fix to address the issue that when topping up a digital wallet, users saw an error when trying to top up over the allowed limit of, say $50. We made a fix so that whenever you entered an amount larger than this number, the textbox would correct itself to the maximum amount of $50. We tested the fix, then started to roll out.

On rollout, we saw a statistically significant drop on top ups happening, so we rolled back. It turned out that people wanting to top up larger amounts were being confused about what was happening when they entered a sum larger than $50, and they abandoned topping up. Had it not been for experimenting, we would have missed the fact that the fix actually caused a regression.

The question of who owns experiments, and coordinating them, is something that should be clarified at the team-level. The most common setup is for either data scientists, or for PMs to own experimentation, working with engineering to make sure the experiment is implemented correctly. However, engineers rarely own the rollout of the control / treatment groups, or decide on how to bucket users.

Further reading on experimentation, much of this is not only applicable to mobile:

- Building an intelligent experimentation platform, analyzing experiment outcomes and a Bring your own metrics platform from Uber
- Experimentation analysis platform from Doordash
- Our new experimentation platform from Spotify
- Our experimentation engine from LinkedIn
- Our experimentation platform, part 1 from Zalando
- Scaling our experimentation platform from Airbnb
- It's about A/B testing: our experimentation platform from Netflix
- Scalable feature toggles and A/B testing from Grab
- Wasabi: an open source A/B testing platform from Intuit
- Building our A/B testing platform from Pinterest
- Experimentation: technical overview from Twitter
- A/B testing mobile apps on iOS from Farfetch
- Building a Culture of Experimentation from Harvard Business Review

35. Advanced Code Quality Checks

The shorter the feedback cycle between detecting issues with your code, the more productive both engineers and teams will be. While getting feedback on your code at code review is great, would it not be even better to get instant feedback, even before you submit your code to code review?

Productive teams put advanced code checking infrastructure in place early on, exactly to help with rapid feedback on easy-to-spot code quality issues. Linting and static analysis are the two most common approaches and are often used together.

Code formatting is one of the most common use cases of code quality checks. The code formatter would be run before creating a pull request, ensuring that all code up for review follows the style guide that the team agreed on and defined in the formatter. Popular code formatters include SwiftFormat or Google-java-format.

Linting is a special case of static analysis; scanning the code for potential errors, beyond just code formatting. This can be as trivial as checks ensuring that indentation is correct, through enforcing naming patterns, all the way to more advanced rules such as declaring variables in alphabetical order. Popular linting tools include:

- iOS: the Clang analyzer — shipping with Xcode — and SwiftLint.
- Android: the lint tool — shipping with Android Studio — and ktlint for Kotlin.

As the team grows, it can make sense to start enforcing more complex rules across the codebase. These rules could be enforcing team-wide coding patterns, such as restricting forced values, or enforcing architecture “rules”, such as a View not being allowed to invoke Interactors directly.

At Uber, we saw lots of value in adding architecture definitions as “lint enforceable” rules. To do so, the team built and open-sourced NEAL (Not Exactly A Linter) for more advanced pattern detection, used across iOS and Android.

Lint fatigue is a problem that starts to occur at large projects, or ones with many linting rules. As the errors or warnings pile up, engineers often start to ignore them. A good example is ignoring deprecation warnings when it is not clear how to migrate to a new implementation of an API.

A common way of dealing with lint fatigue is to make linting errors break the build, leaving no choice but to fix them. It is a bit annoying, but effective. Another approach is to build tools to fix linting errors automatically. This is an approach Instagram took; they used automated refactoring to educate engineers about coding best practices.

Static analysis is the more generic phrase of automatic inspection of the code, looking for potential issues and errors. Mobile static analysis tools usually help to detect use cases that are more complex than what a simple lint rule could catch.

Most static analysis tools are written for a language — Swift, Kotlin, Objective C, or Java — and detect common programming issues such as unused variables, empty catch blocks, possible null values, and others. On top of the linting tools listed above, static analysis tools you could consider are:

- Swift, Kotlin, Objective-C and Java: SonarQube and SonarCloud (advanced static analysis), NEAL, Infer (Java, Objective-C)
- Swift: Clang analyzer (ships with Xcode), SwiftLint, SwiftInfo, Tailor, SwiftFormat
- Kotlin: ktlint (a “no-decision” linter), detekt (code smells and complexity reports)
- Objective-C: Clang analyzer (ships with Xcode), OCLint
- Java: lint (ships with Android Studio), NullAway (annotation-based null-checks), FlowDroid (data flow analysis), CogniCrypt (secure cryptography integration checks) PMD (Programming Mistake Detector), Checkstyle
- See also this repository of static analysis tools per language

The upside of using linting and static analysis tools is getting more rapid feedback and code reviewers not needing to check for the common code issues. Code quality generally stays higher with the tools enforcing rules. When using advanced tooling, static analysis can result in more stable and secure apps by detecting edge cases ahead of time. A good example of added stability is using a tool to prevent runtime crashes due to null objects and doing this by analyzing the code, compile-time.

The downside of these tools is the time it takes to integrate them and the additional maintenance they bring. You need to decide which tool to use and add them to your build setup, both to local builds and the CI/CD setup. Once in place, you might need to keep the rules up to date, and every now and then, update the version of the tool to support new features you might need.

The more complex tooling you choose, the more this maintenance might add up. At Uber, we set up extensive linting and static analytic checks. The outcome felt to me like it was worth the added effort. However, I would be hesitant to build the type of custom tooling we did, and instead would use a good enough tool for the job that can be set up with little effort.

Code coverage—checking which parts of your code are tested via unit tests, or other automated tests—is another tool to help with quality. For iOS, Xcode has built-in code coverage reports. For Android, Jacoco is a commonly used coverage reporting tool for Java and Kotlin. Several vendors also offer code coverage solutions. By integrating code coverage with your development workflow and CI setup, you can:

- Visualize how much coverage each change has. This can make it easy for engineers and code reviewers to verify if new business logic has been tested.
- Help enforce a minimum code coverage policy, if the team has agreed on an approach like this.
- Show code coverage trends over time. Do code coverage changes have any correlation with outages or other issues with the app? You will have the data to answer this question.
- Identify areas that are poorly covered, and where adding tests could help with correctness, maintainability and give more confidence to engineers making changes in those areas.

Further reading:

- Static analysis at scale from Instagram
- Kotlin linting with ktlint from Pinterest
- iOS linting tooling from Pinterest
- iOS: Continuous code inspection with SwiftLint & SwiftLint in Use
- Android: Android lint overview & Android lint framework - an introduction