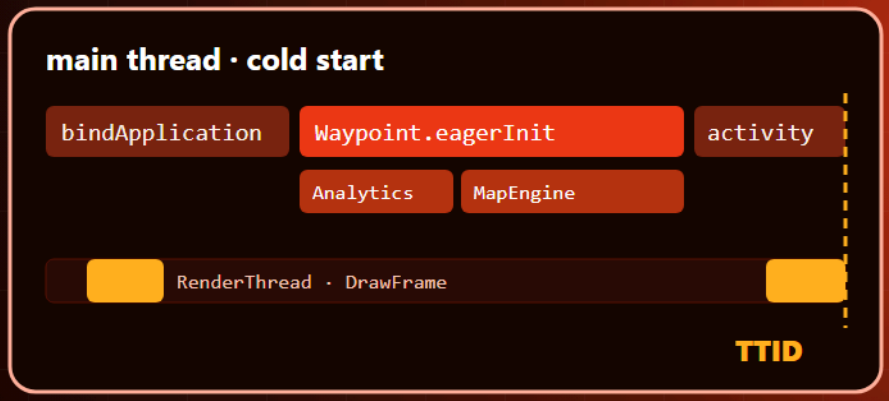
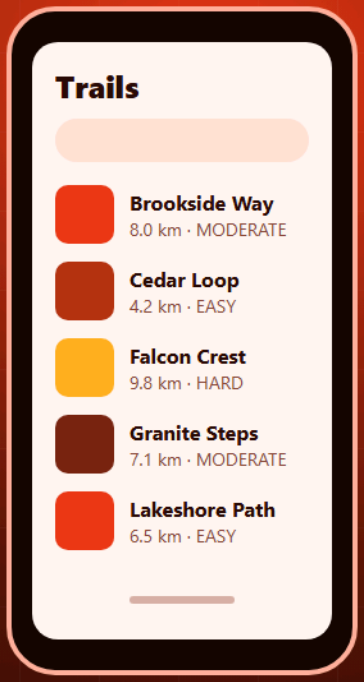
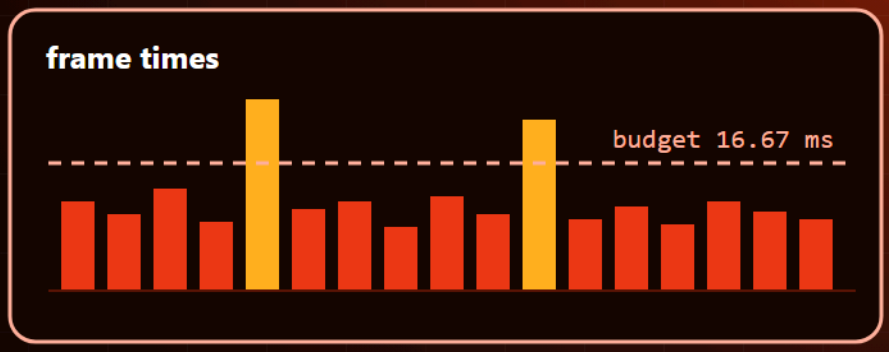







```
rule.measureRepeated(
  metrics = listOf(StartupTimingMetric()),
  compilationMode = CompilationMode.Partial(Require)) { }
```

MEASURE WHAT USERS FEEL

Android Performance and App Quality

Startup, rendering, memory, battery, and app size — measured and fixed

Baseline Profiles · Macrobenchmark · Perfetto · Compose stability
JankStats · StrictMode · LeakCanary · R8 · App Bundles

STARTUP · FRAMES · MEMORY · BATTERY · SIZE

THENEXBOOKS

Android Performance and App Quality

Startup, Rendering, Memory, Battery, App Size, and Baseline Profiles

Measure what users feel, fix what matters, and keep it fast in production

First Edition

THENEXBOOKS

Android Performance and App Quality
Startup, Rendering, Memory, Battery, App Size, and Baseline Profiles — Measuring What Users Feel

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Android, Android Studio, the Android Gradle Plugin, Jetpack libraries, Google Play, and the Play Console change frequently; tool windows, metric names, thresholds, and APIs can differ between versions. Performance results depend on the device, the Android version, and the build; always measure on your own devices and consult the official documentation for the versions you use.

Android, Google Play, the Google Play Console, Android Studio, Jetpack Compose, and Firebase are trademarks of Google LLC. Kotlin is a trademark of the Kotlin Foundation. Perfetto, LeakCanary, OkHttp, Retrofit, Coil, and other named open-source projects belong to their respective owners. Other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by Google LLC or any other company mentioned.

Northbeam Labs, the Waypoint app, its team members, users, trails, trips, and all sample data used as the running example in this book are fictional. The backend address api.waypoint.example uses a reserved example domain, and no real service is required to follow along.

Preface

Users rarely say "this app has a slow time to initial display." They say it feels sluggish, that the list stutters when they scroll, that their battery drained on a hike, or that the download was too big for their data plan. Then they leave a one-star review, or they simply stop opening the app. Performance is not a polish step at the end of a project; it is part of what an app *is*, as much as its features and its design.

The good news is that Android performance work is no longer guesswork. Modern Android gives us precise instruments: system traces and Perfetto, the Android Studio profilers, Macrobenchmark and Microbenchmark, Baseline Profiles, JankStats, Compose compiler reports, R8 and APK analysis, StrictMode, LeakCanary, and the field data in Android vitals. Used well, these tools turn "it feels slow" into a specific, measurable problem with a specific fix — and they let you prove that the fix worked and stays fixed.

This book teaches that discipline end to end. We do not collect random tips. We take one realistic app, measure it, find what users actually feel, fix it with the right tool, verify the result, and finally write a performance budget that keeps the app fast release after release.

Who This Book Is For

- **Android developers** who can build an app with Kotlin and Jetpack Compose and now want it to start fast, scroll smoothly, and stay stable on every device.
- **Tech leads and senior engineers** who need a repeatable way to measure, budget, and protect performance across a team and a release train.
- **QA and release engineers** who want to understand Android vitals, benchmarks, and performance regressions well enough to catch them before users do.
- **Developers maintaining older apps** who face slow startup, ANRs, memory leaks, or a growing download size and need a systematic way out.

What You Will Learn

- How to think about performance from the user's side, and what Android vitals and the Play Console really measure.
- How Android runs your app — processes, ART compilation, and the frame pipeline — so the measurements make sense.
- How to trace and profile with Perfetto and the Android Studio profilers, and how to benchmark honestly with Macrobenchmark and Microbenchmark.
- How to make startup fast: lazy initialization, the App Startup library, Baseline Profiles, Startup Profiles, profileinstaller, and a first screen that is useful immediately.

- How to keep rendering smooth: jank and frame metrics, JankStats, Compose stability and skipping, lazy list keys, derivedStateOf, and image loading.
- How to keep the app responsive and lean: ANRs, StrictMode, dispatchers, memory, leaks with LeakCanary, and fast database access.
- How to save battery, data, and storage: location and background work, HTTP caching, R8, resource shrinking, and App Bundles.
- How to keep it all fast in production: monitoring, stability and exit reasons, regression testing in CI, device tiers, and a performance budget.

How This Book Is Organized

Part I — Measuring What Users Feel (Chapters 1–5) sets up the mindset, Android vitals, how Android runs your app, tracing and profiling, and benchmarking. **Part II — Fast Startup** (Chapters 6–10) covers startup types and metrics, lazy initialization and the App Startup library, Baseline Profiles, Startup Profiles and profileinstaller, and the first screen. **Part III — Smooth Rendering** (Chapters 11–15) covers jank, frame metrics, Compose performance, lazy lists and state, and images. **Part IV — Responsiveness and Memory** (Chapters 16–19) covers the main thread and ANRs, memory, leaks, and data access. **Part V — Battery, Network, and Size** (Chapters 20–23) covers battery, network, R8, resources, and App Bundles. **Part VI — Quality in Production** (Chapters 24–28) covers production monitoring, stability, CI regression testing, device diversity, and a complete performance budget. The appendices provide a cheat sheet, templates and reference tables, a production checklist, and a glossary.

Each chapter ends with "Try It Yourself" exercises and a **Key Takeaways** box for quick review.

The Running Example: Waypoint

Throughout the book we tune **Waypoint**, a hiking trip planner and tracker built by **Northbeam Labs**, a small fictional product studio. Waypoint lets hikers browse trails such as Cedar Loop and Falcon Crest Trail, plan trips, record hikes with location tracking, and keep a private journal. Lena Ortiz leads Android development, Sam Okafor runs the backend, Priya Natarajan owns product and design, and Theo Brandt leads QA. Our starting point is a working Waypoint build with typical real-world problems: too much work in `Application.onCreate()`, a trail list that recomposes and scrolls more than it should, no Baseline Profile, no HTTP cache, and resource shrinking switched off. Chapter by chapter we measure each problem, fix it, and verify the fix, and in Chapter 28 we write the performance budget that keeps Waypoint fast. Waypoint is also the running example in other books of the thenexbooks Android series, but this book is completely self-contained.

Versions and Conventions

Examples use **Kotlin 2.4**, **Jetpack Compose** (Compose BOM 2026.09.00), the **Android Gradle Plugin 9.4** with **Gradle 9.8**, **compileSdk 37**, **targetSdk 36**, and **minSdk 26**, together with the Jetpack benchmark libraries 1.5 and profileinstaller 1.4 — the stable releases at the time of writing. Newer versions usually work the same way; where behavior depends on the Android version or a library version, a Note says so. Play Console pages, metric names, and thresholds change over time, so we describe what each metric means and tell you where to look rather than quoting numbers that may be outdated.

A word about numbers. Performance results depend on the device, its Android version, its thermal state, and the build being measured. Whenever this book shows benchmark or profiler output that comes from a device, it is labeled as an example of the *shape* of the output — your numbers will differ. Numbers that come from deterministic tools, such as an APK size breakdown or a Compose compiler report, were produced by running those tools on the book's sample project.

> **TIP**
>
> A recommended practice or a technique that makes your work easier.

> **NOTE**
>
> Background information, version differences, or a useful aside.

> **WARNING**
>
> A common mistake or something that can make performance, stability, or measurements worse.

Code, Gradle tasks, and commands appear in `monospace`; long lines are wrapped to fit the page. Android Studio menu paths are written in bold with arrows, such as **View > Tool Windows > Profiler**. Waypoint's modules, classes, and functions use the same names in every chapter; Appendix B lists them all.

Contents

Preface

PART I — MEASURING WHAT USERS FEEL

1 Performance Is a Feature: What Users Feel and the Waypoint Starter

2 Android Vitals and Play Console Quality Signals

3 Under the Hood: Processes, ART Compilation, and the Frame Pipeline

4 Tracing and Profiling: Perfetto, System Traces, and Android Studio Profilers

5 Benchmarking Done Right: Macrobenchmark, Microbenchmark, and Honest Numbers

PART II — FAST STARTUP

6 App Startup Explained: Cold, Warm, and Hot Starts, TTID, and TTFD

7 Doing Less at Launch: Lazy Initialization and the App Startup Library

8 Baseline Profiles: Generating, Shipping, and Verifying

9 Startup Profiles, DEX Layout, and profileinstaller

10 The First Screen: Splash Screens, reportFullyDrawn, and Loading Startup Data

PART III — SMOOTH RENDERING

11 Jank Explained: Frames, Deadlines, and Where the Time Goes

12 Measuring Smoothness: JankStats, FrameTimingMetric, and Scroll Benchmarks

13 Compose Performance: Recomposition, Stability, and Skipping

14 Compose Lists and State: Keys, contentType, derivedStateOf, and Deferred Reads

15 Image Performance: Decoding, Sizing, Caching, and Coil

PART IV — RESPONSIVENESS AND MEMORY

16 Keeping the Main Thread Free: ANRs, StrictMode, and Dispatchers

17 Memory Fundamentals: The Heap, Garbage Collection, and Memory Pressure

18 Finding and Fixing Memory Leaks with LeakCanary and Heap Dumps

19 Fast Data: Room Queries, Indexes, Paging, and Disk I/O

PART V — BATTERY, NETWORK, AND SIZE

20 Battery Efficiency: Location, Wake Locks, and Background Work

21 Network Efficiency: Caching, Compression, Batching, and Prefetching

22 Shrinking Code with R8: Optimization, Keep Rules, and Mapping Files

23 Resources, Native Code, and App Bundles: Measuring What Users Download

PART VI — QUALITY IN PRODUCTION

24 Monitoring in Production: Vitals, Custom Traces, and Field Metrics

25 Stability as Quality: Crashes, ANRs, and Exit Reasons

26 Performance Regression Testing in CI

27 Performance Across Devices: Low-End Phones, Device Tiers, and Android Versions

28 Case Study: A Performance Budget for Waypoint

APPENDICES

A Android Performance Cheat Sheet

B Templates and Reference

C Production Checklist

D Glossary

PART I

Measuring What Users Feel

Why performance is a feature, what Android vitals and the Play Console actually report, how Android runs your code from process start to the frame on screen, and the measuring tools of the trade: system traces, Perfetto, the Android Studio profilers, Macrobenchmark, and Microbenchmark — used honestly.

1 · Performance Is a Feature

2 · Android Vitals and Play Console Quality Signals

3 · Under the Hood: Processes, ART, and Frames

4 · Tracing and Profiling

5 · Benchmarking Done Right

CHAPTER 1 · PART I — MEASURING WHAT USERS FEEL

Performance Is a Feature: What Users Feel and the Waypoint Starter

Users never see a frame time or a heap size; they feel an app that opens quickly or slowly, scrolls smoothly or stutters, and drains their battery or does not. In this chapter we connect those feelings to things we can measure, introduce the measure → find → fix → verify loop that drives the whole book, and build the deliberately slow Waypoint starter that we will tune chapter by chapter.

What Users Feel

Ask someone why they stopped using an app and you will rarely hear "the 90th percentile frame duration was too high". You will hear "it took forever to open", "it kept freezing", "it ate my battery", or "it crashed when I needed it". These complaints are the raw material of performance work. Our job is to translate each of them into something a tool can measure, because only measured problems can be fixed with confidence and only measured fixes can be trusted not to regress.

It helps to sort what users feel into a small number of dimensions. Each dimension has a natural question the user asks, a set of metrics we can collect, and a part of this book that deals with it.

Dimension	The user's question	What we measure	Part
Launch	Why is it still loading?	Cold/warm/hot start, TTID, TTFD	II
Smoothness	Why does it stutter?	Frame duration, frame overrun, jank	III
Responsiveness	Did it hear my tap?	Main-thread blocking, ANRs	IV
Memory	Why did it restart?	Heap use, leaks, process kills	IV
Battery	Why is my phone warm?	Wakeups, location use, radio time	V
Data and storage	Why is it so big?	Download size, bytes transferred	V
Stability	Why did it close?	Crash rate, ANR rate, exit reasons	VI

Notice that stability sits in the same table as speed. That is intentional. A crash is the ultimate performance failure: the app took infinitely long to do what the user asked. An ANR is a performance bug that the system has escalated into a stability event. Throughout the book we treat performance and stability as one subject, "app quality", because users experience them as one.

Notice also that every row mixes something the user can see directly (a loading screen, a stutter) with something they only notice indirectly (a warm phone, a full storage warning). The indirect ones are easy to ignore during development because a developer's phone is usually fast, charged, and plugged into a desk. Users carry their phones up mountains. Waypoint is a hiking app, so its users care about battery and offline behavior far more than the developer testing it on office Wi-Fi.

Perceived performance

What users feel is not identical to what a stopwatch measures. Two screens that take the same time to load can feel very different. One shows a blank white surface until everything is ready; the other shows the app's structure immediately, fills in the trail names from the local database, and lets the photos arrive a moment later. The second one feels faster because the user can start reading, scrolling, and deciding while the rest loads.

This is **perceived performance**, and it is a legitimate engineering goal, not a trick. It rests on a few principles that we will apply repeatedly:

- **Show something useful early.** Render the first screen from local data before the network answers. Waypoint is offline-first: the UI reads Room, and network refreshes update Room in the background.
- **Respond to input immediately.** A tap that shows a pressed state or a ripple right away feels responsive even if the next screen takes a moment. A tap that does nothing for half a second feels broken.
- **Keep motion smooth.** Users forgive a short wait more readily than a stuttering scroll, because stutter feels like the app is struggling.
- **Be honest about progress.** Skeleton rows and placeholders tell the user what is coming. An indefinite spinner tells them nothing.

Perceived performance also explains why this book distinguishes between two startup metrics that we define fully in Chapter 6. **Time to initial display** (TTID) is when the first frame appears. **Time to full display** (TTFD) is when the screen is actually useful, which the app reports itself. A fast TTID with a slow TTFD means the user sees a screen quickly but cannot use it; the starter does not report TTFD at all (PERF-10), which is one of the first gaps we close.

The cost of slowness

You will find plenty of figures online claiming that each extra second of loading costs a certain percentage of users or revenue. We do not repeat them here, because they come from specific products, markets, and years, and they rarely transfer to your app. What we can say with confidence is how slowness hurts, even without a number attached:

- **Users leave.** An app that is slow to open gets opened less. On a trail with one bar of signal and cold fingers, a hiker who waits several seconds for Waypoint will reach for a paper map next time.

- **Users complain in public.** Reviews mention "slow", "laggy", and "battery" far more readily than they praise speed. Good performance is mostly invisible; bad performance is quoted.
- **Play notices.** Google Play collects crash, ANR, startup, rendering, and battery data through Android vitals and can reduce an app's visibility when its behavior is bad enough. Chapter 2 covers this in detail.
- **The team slows down.** Performance problems that nobody measures pile up as debt. Each new feature makes the cold start a little longer, until nobody remembers when it was fast or which change made it slow.

The good news is the inverse of the last point. A team that measures continuously can keep performance roughly constant while the app grows, because every regression is caught at the commit that introduced it. That is the destination of this book: Chapter 26 automates the checks and Chapter 28 writes them down as a budget.

Measure → Find → Fix → Verify

Every chapter in this book follows the same loop, and every chapter says which tool proves the result. It is worth stating the loop carefully now, because skipping a step is the most common way performance work goes wrong.

1. **Measure.** Turn a feeling ("the list stutters") into a number from a repeatable measurement ("the 90th-percentile frame overrun while flinging the trail list on our reference phone"). Without a number, you cannot tell whether a fix helped.
2. **Find.** Locate the cause, usually with a trace or a profiler. The number says *how much*; the trace says *where*. Resist the temptation to guess. Performance intuition is wrong surprisingly often, even for experienced developers.
3. **Fix.** Change the smallest thing that addresses the cause. Make one change at a time, so that the next step can attribute the result.
4. **Verify.** Repeat the same measurement under the same conditions and compare. If you cannot show a difference that is larger than the measurement's own noise, you have not shown an improvement, however reasonable the change looks.

Here is how the loop looks for the first problem we will fix in Waypoint. You do not need to understand the tools yet; the point is the shape.

Step	Waypoint example (PERF-1)	Tool
Measure	Cold start TTID on the reference phone	StartupBenchmark (Ch. 5, 6)
Find	Wide `Waypoint.eagerInit` section on the main thread	System trace (Ch. 4)
Fix	Lazy SDK init, work moved off the main thread	App Startup library (Ch. 7)
Verify	Same benchmark, same device, compare distributions	StartupBenchmark (Ch. 7)

> **TIP**
>
> Before you change any code for performance, write down three things: the metric, the current value with its spread, and the tool that produced it. If you cannot fill in all three, you are still in the measure step, not the fix step.

Lab data and field data

Measurements come from two very different places, and good teams use both.

Lab data is what you measure yourself under controlled conditions: a benchmark on a known device, a trace you recorded, a size report from a build. Lab data is precise, repeatable, and fast to obtain. You can run it on every commit. Its weakness is that it represents only the devices, data, and journeys you chose. Your reference phone is not your users' phones, and your test account with eight trails is not a power user with hundreds of saved trips.

Field data is what happens on real users' devices: Android vitals in Play Console, and any metrics your app reports itself. Field data is the truth about what users feel, across every device model and network. Its weaknesses are delay (days, not minutes), coarseness (aggregates, not traces), and noise (real life is messy).

The two complement each other. Field data tells you which problems matter and confirms that a fix reached users. Lab data finds causes and proves fixes before they ship. In the loop above, field data is usually where a problem is first noticed and where it is finally confirmed as solved; lab data does the work in between.

> **NOTE**
>
> One rule applies to every number in this book: we never invent measurements. When a tool can run on the book's build machine (a Windows computer with the Android SDK but no emulator or phone), we quote its real output and say so. When a tool needs a device, such as Macrobenchmark or Perfetto, we show the *shape* of its output with placeholders like `###.#` instead of digits. Run the same tools on your own device to fill them in.

Meet Northbeam Labs and Waypoint

Our running example is **Waypoint**, a hiking trip planner and tracker built by **Northbeam Labs**, a small product studio. Waypoint lets people browse and search trails, read trail details, plan trips, and record hikes. In this book we work on a focused slice of it: the trail list and trail detail screens, the local database and network layer behind them, and later the hike recording path, because together they exercise every performance dimension in the table above.

Four people at Northbeam appear throughout the book, each with a clear role in performance work:

- **Lena Ortiz**, the Android lead, usually drives the fixes: startup changes, Compose tuning, memory work.
- **Sam Okafor** owns the backend. When the answer is a cache header or a batching endpoint, it is Sam's change.
- **Priya Natarajan** owns product and design. She speaks for what users feel and signs off on the performance budget.
- **Theo Brandt** runs QA: benchmarks, the device lab, and the CI checks that keep fixes from regressing.

The split matters because performance is a team sport. A startup regression might come from a new SDK (Lena), a slow first response (Sam), a design that asks for too much on the first screen (Priya), or a benchmark that stopped running (Theo). When we finish in Chapter 28, each line of Waypoint's budget has an owner.

The Waypoint Performance Starter

The book's sample is a small but realistic multi-module project in the folder `sample\waypoint` of the book's download. It is called the **performance starter** because it is deliberately not tuned: it contains ten planted performance problems that we find and fix over the course of the book. It builds with the toolchain that was current at the time of writing: Kotlin 2.4.20, Android Gradle Plugin 9.4.1 with built-in Kotlin support, Gradle 9.8.0, JDK 21, compileSdk 37, targetSdk 36, minSdk 26 (28 for the benchmark module), and the Compose BOM 2026.09.00. Recent versions of these tools should work with at most small adjustments.

Module tour

Module	What lives there
`:app`	`WaypointApp`, `MainActivity`, navigation routes, and `startup/FakeSdks.kt` (two simulated third-party SDKs)
`:core:model`	Pure Kotlin/JVM models (`Trail`, `Trip`, `TrackPoint`), `SampleTrails`, `SyntheticTrails`, `TrackMath`, `TrailFormat`
`:core:network`	Retrofit `WaypointApi` for the fictional `api.waypoint.example` backend
`:core:database`	Room: `TrailEntity`, `TrailDao`, `WaypointDatabase` ("waypoint.db")
`:core:data`	`TrailRepository` and its offline-first implementation
`:feature:trails`	Trail list and detail screens (Compose) and their ViewModels
`:benchmark`	Macrobenchmarks and the Baseline Profile generator (needs a device)
`:microbenchmark`	Jetpack Microbenchmark for `TrackMath` (needs a device)

The dependencies flow in one direction, from features down to the core modules. The two benchmark modules sit beside the app: `:benchmark` drives the installed app from the outside, like a user would, while `:microbenchmark` measures a single function in isolation.

```
          :app ------------------------------+
           |                                 |  drives the installed app
           v                                 |  (UiAutomator, no code access)
   :feature:trails                    :benchmark
           |
           v
      :core:data
       |      |
       v      v
:core:network  :core:database
       \      /
        v    v
      :core:model  <---- :microbenchmark (TrackMath only)
```

A few details of the starter matter for performance work from the first day:

- The trail list exposes Compose test tags such as `trail_list` and `trail_row_tr-005` as resource IDs (`testTagsAsResourceId` in `MainActivity`), so UiAutomator-based benchmarks can find them.
- Starting `MainActivity` with the extra `waypoint.stressTrails` set to 500 adds 500 deterministic practice trails, so the list holds 508 trails and is long enough to scroll in benchmarks.
- Custom trace sections already exist: `Waypoint.eagerInit`, `FakeAnalytics.init`, and `FakeMapEngine.warmUp`. Chapter 4 shows how they appear in a system trace.
- The manifest declares `<profileable android:shell="true">`, which lets profilers and benchmarks inspect release builds.

The simulated SDKs

Real apps start slowly for many reasons, but a very common one is third-party SDK initialization in `Application.onCreate()`. Rather than depend on a real analytics or map SDK (whose cost we could not control or reproduce), the starter contains two stand-ins with fixed costs:

app/src/main/kotlin/com/northbeam/waypoint/startup/FakeSdks.kt (excerpt)

```
object FakeAnalytics {
    const val INIT_COST_MS = 150L
    fun init(context: Context) = trace("FakeAnalytics.init") {
        if (isInitialized) return@trace
        Thread.sleep(INIT_COST_MS)
        isInitialized = true
    }
}

object FakeMapEngine {
```

```
    const val WARM_UP_COST_MS = 250L
    fun warmUp(context: Context) = trace("FakeMapEngine.warmUp") { /* sleeps 250 ms */ }
}
```

Together they put 400 ms of deliberate sleep on the main thread during every cold start. These are teaching props, not measurements of any real SDK. Their value is that they make the cost visible and deterministic: when we remove them from the startup path in Chapter 7, we know exactly how much simulated blocking work disappeared.

A cold start through the hiker's eyes

Let us walk through what happens when a hiker taps the Waypoint icon and the app's process is not running. We cover the platform details in Chapter 3 and the metrics in Chapter 6; for now, focus on the order of events and on what the user sees at each moment.

```
tap icon
  |
  |  system creates the process, loads the app's code
  |  WaypointApp.onCreate()                          main thread
  |    FakeAnalytics.init ........ 150 ms sleep      blocked
  |    FakeMapEngine.warmUp ...... 250 ms sleep      blocked
  |    runBlocking { seedIfEmpty() }  disk I/O       blocked
  |  MainActivity.onCreate(), setContent { ... }
  |  first frame drawn ............................. TTID (initial display)
  |    trail list shows "loading" until Room emits
  |  trail rows on screen, photos loading .......... TTFD (not reported yet)
  v
```

During everything above the "first frame" line, the user sees only the system's starting window (on Android 12 and later, a system splash screen with the app icon). They cannot do anything. The 400 ms of sleep plus the database wait all happen in that window, before Waypoint has drawn a single pixel of its own. Only the two sleeps have a known duration; how long the database seed takes depends on the device and its storage, which is exactly why we measure it rather than guess.

Two things in this timeline deserve attention now, because they set up the next several chapters.

First, **nothing in `onCreate()` is needed to draw the first frame**. The trail list reads Room through a `Flow` and already shows a loading state while it waits, so the database seed does not need to block. The analytics SDK can start later. The map engine is only used on the trail detail screen, which the user may never open in this session. This is the typical shape of startup debt: work that is placed at launch because `Application.onCreate()` is a convenient place to put things, not because the first frame needs it.

Second, **the end of the timeline has no marker**. The app never tells the system when the trail list is actually usable, so no tool can report the time to full display. That is PERF-10, and it is why we fix the reporting (Chapter 10) as part of the startup work rather than as an afterthought: without TTFD,

we could make the first frame appear sooner while making the useful screen appear later, and our startup benchmark would call that a win.

Keep this timeline in mind. In Chapter 4 we will see the same sequence as colored slices in a system trace, with the `Waypoint.eagerInit` section sitting on the main thread before the first frame.

Build and Test the Starter

Before we change anything, we build the starter and run its tests. This gives us a known-good baseline and checks that your toolchain matches. Copy the starter somewhere you can modify it (never edit the original, so you can always compare with it), then run Gradle from the copy:

Terminal

```
./gradlew test assembleDebug :app:assembleRelease
```

On Windows, use `gradlew.bat` with the same arguments. The first run downloads dependencies and can take a while; later runs use Gradle's caches. On the book's build machine, with a warm cache, the end of the output looked like this (your timing and task counts will differ):

Output (real, book build machine)

```
> Task :app:lintVitalRelease
> Task :app:packageRelease
> Task :app:assembleRelease
> Task :app:createReleaseApkListingFileRedirect

BUILD SUCCESSFUL in 19s
474 actionable tasks: 229 executed, 245 from cache
```

The `test` task runs 16 JVM tests in 9 classes. Several of them use Robolectric, which runs Android code such as Room and Compose UI tests on the desktop JVM without a device. These tests are not performance tests; they are the safety net that lets us optimize aggressively without breaking behavior.

Test class	Module	Tests
`SampleTrailsTest`	`:core:model`	3
`TrackMathTest`	`:core:model`	3
`TrackMathTimingTest`	`:core:model`	1
`WaypointApiTest`	`:core:network`	1
`TrailDaoTest`	`:core:database`	1
`OfflineFirstTrailRepositoryTest`	`:core:data`	3
`TrailListViewModelTest`	`:feature:trails`	1
`TrailListScreenTest`	`:feature:trails`	2

`MainActivityTest`	`:app`	1

The benchmark modules compile as part of a full build, but their tests do not run here: Macrobenchmark and Microbenchmark need a physical device (Chapter 5 explains why emulators are not good enough).

> **NOTE**
>
> `TrackMathTimingTest` is a "microbenchmark-style" timing test on the desktop JVM. It prints a median time for computing the distance of a 10,000-point synthetic hike. Its numbers depend on your computer and say nothing about phones; Chapter 5 uses it to show why naive timing is noisy.

Debug Versus Release: Never Judge Performance on Debug

The single most common performance mistake on Android is measuring a debug build. It is understandable: debug is what Android Studio installs when you press Run. But a debug build is a different program from the one your users run, and its performance tells you very little about theirs.

Look at the starter's release configuration. R8 is on (`isMinifyEnabled = true`), which shrinks, optimizes, and obfuscates the code. Resource shrinking is still off; that is planted problem PERF-9.

app/build.gradle.kts (starter, excerpt)

```
buildTypes {
    release {
        isMinifyEnabled = true
        // STARTER STATE: resource shrinking is OFF (Chapters 22-23).
        isShrinkResources = false
        proguardFiles(
            getDefaultProguardFile("proguard-android-optimize.txt"),
            "proguard-rules.pro",
        )
        // Debug key so release/benchmark builds can be installed locally.
        signingConfig = signingConfigs.getByName("debug")
    }
}
```

Now compare the two APKs that the build just produced. We used `apkanalyzer` from the Android SDK command-line tools (Chapter 23 explains it in depth):

Terminal

```
apkanalyzer apk file-size app/build/outputs/apk/debug/app-debug.apk
apkanalyzer apk file-size app/build/outputs/apk/release/app-release.apk
apkanalyzer files list app/build/outputs/apk/debug/app-debug.apk
```

```
Output (real, book build machine, these exact versions)
```

Build	APK file size	DEX files	Code optimized?
Debug	14,285,025 bytes	15	No
Release (R8)	2,090,919 bytes	1	Yes

The release APK is less than a sixth of the debug APK's size, and its code fits in a single DEX file instead of fifteen. Those byte counts are valid only for this exact build and these library versions, but the pattern holds for any app. Size is the visible part; the invisible part matters more for speed:

- **Debuggable apps run differently.** When an app is marked debuggable, ART keeps debugging support enabled and gives up many of the optimizations it would otherwise apply, so code typically runs noticeably slower than in a release build.
- **No R8 optimizations.** Debug builds skip inlining, class merging, and dead-code removal.
- **Debug-only tooling.** Compose tooling, inspection support, and libraries such as LeakCanary (Chapter 18) add work that users never run.
- **Different compilation state.** Baseline Profiles (Chapter 8) are about how ART compiles release code; a debug build tells you nothing about whether they work.

> **WARNING**
>
> Never report a startup time, frame time, or memory figure measured on a debug build, and never compare a debug measurement with a release one. Use a release build (or the non-debuggable `benchmark` build type that the Baseline Profile plugin creates) with `<profileable android:shell="true">`, as the starter already does. Debug builds are for debugging correctness, not for judging speed.

This does not mean debug builds are useless for performance work. Some tools are deliberately debug-only: StrictMode (Chapter 16) flags disk and network access on the main thread, and LeakCanary (Chapter 18) finds memory leaks. Those tools detect *patterns* that are slow on any build. What they do not give you is a trustworthy *time*.

The Planted Performance Debt

The starter contains ten deliberate problems. Each has an ID that we use throughout the book, so you can always tell which fix a chapter is about. Read the list now; by the end of the book each line will have a measured before, a fix, and a verified after.

ID	Problem in the starter	Fixed in
PERF-1	`WaypointApp.onCreate()` runs `FakeAnalytics.init` (150 ms) and `FakeMapEngine.warmUp` (250 ms) on the main thread	Ch. 7

PERF-2	`runBlocking { seedIfEmpty() }` in `onCreate()`: the main thread waits for disk I/O	Ch. 7 (seen in traces; Ch. 16 explains why StrictMode misses it)
PERF-3	No app Baseline Profile (only library profiles are merged)	Ch. 8 and 9
PERF-4	`Trail` (pure JVM module) is inferred unstable by the Compose compiler	Ch. 13
PERF-5	List filtered inside composition; `items()` without `key` or `contentType`	Ch. 14
PERF-6	Coil uses default settings (no app-wide `ImageLoader`)	Ch. 15
PERF-7	No index on the `region` column of `trails`	Ch. 19
PERF-8	OkHttp has no HTTP cache	Ch. 21
PERF-9	`isShrinkResources = false`	Ch. 23 (code side: Ch. 22)
PERF-10	No `reportFullyDrawn`, so TTFD is unknown	Ch. 10

Each problem is a pattern found in real apps, simplified so you can see it clearly. PERF-1 and PERF-2 are especially common because they start small: one SDK, one quick database check, each "only a few milliseconds" on the developer's phone.

The Road Ahead

The book is organized in six parts that follow the dimensions from the start of this chapter:

- **Part I — Measuring What Users Feel** (Chapters 1–5): Android vitals, how Android runs your code and draws frames, tracing and profiling, and benchmarking you can trust.
- **Part II — Fast Startup** (Chapters 6–10): start types, lazy initialization, Baseline and Startup Profiles, and the first screen.
- **Part III — Smooth Rendering** (Chapters 11–15): jank, frame metrics, Compose stability, lists, and images.
- **Part IV — Responsiveness and Memory** (Chapters 16–19): ANRs and StrictMode, memory, leaks, and fast data access.
- **Part V — Battery, Network, and Size** (Chapters 20–23): location and background work, network efficiency, R8, and app size.
- **Part VI — Quality in Production** (Chapters 24–28): field monitoring, stability, CI regression testing, device diversity, and Waypoint's performance budget.