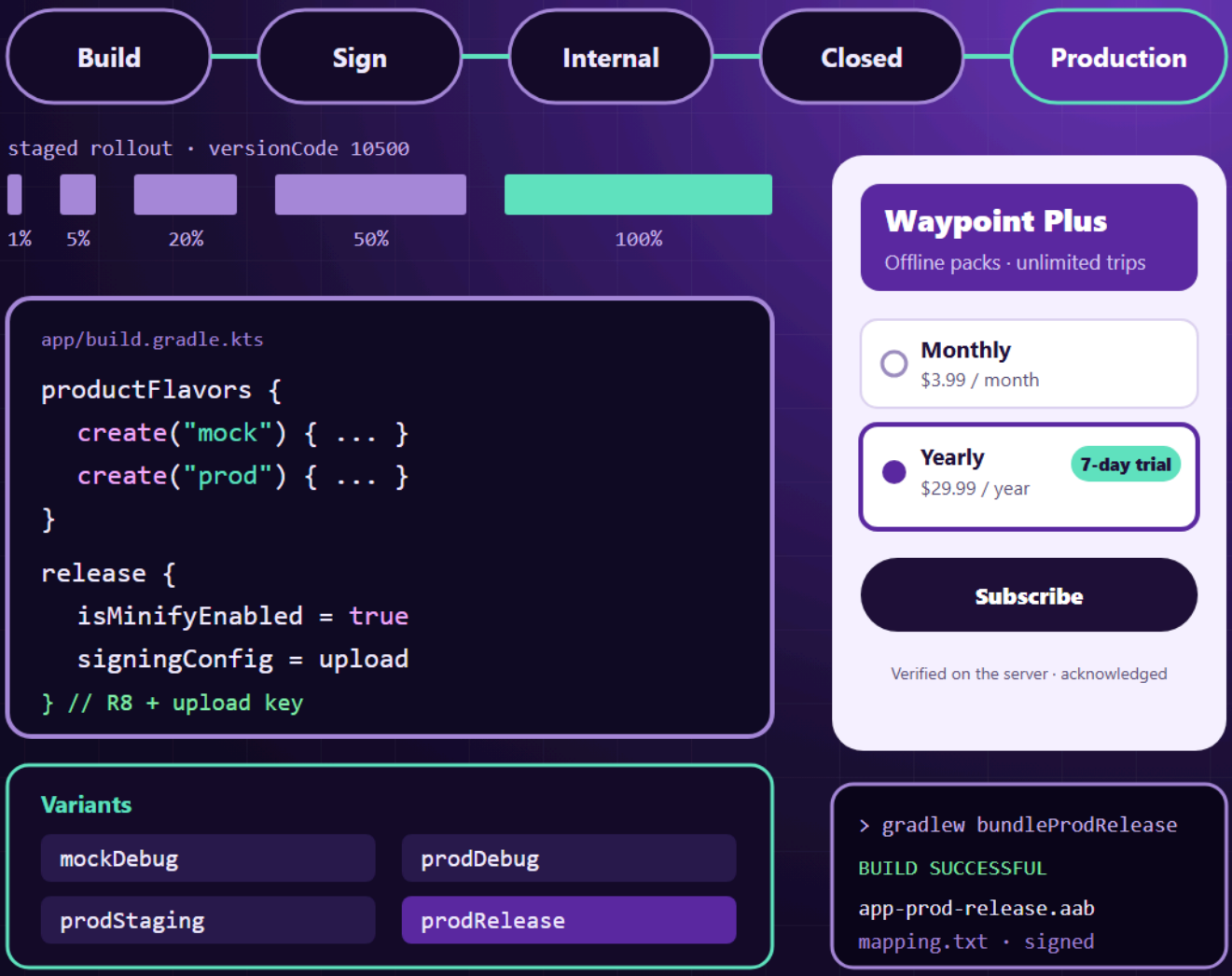


A COMPLETE, PRACTICAL GUIDE

Android Build, Release, and Google Play

From a laptop build to a signed, automated, staged release on Google Play

Gradle · AGP · Version Catalogs · Build Variants
R8 · App Bundles · App Signing · GitHub Actions
Play Console Tracks · Play Billing · Shipping Safely

GRADLE 9 · AGP 9 · KOTLIN 2.4 · JDK 21

THENEXBOOKS

Android Build, Release, and Google Play

Gradle and AGP, Build Variants, Signing, CI/CD, Play Console Tracks, In-App Billing, and Shipping Safely
From a laptop build to a signed, automated, staged release — step by step

First Edition

THENEXBOOKS

Android Build, Release, and Google Play
Gradle and AGP, Build Variants, Signing, CI/CD, Play Console Tracks, In-App Billing, and Shipping Safely

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, build scripts, and workflow files 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. Gradle, the Android Gradle Plugin, Android Studio, Google Play Console, Google Play policies, the Play Billing Library, and GitHub Actions change frequently; menu names, screens, requirements, deadlines, limits, and fees can differ from what is described here. Always check the official documentation and the current Google Play policies before you release, and never rely on this book for legal, tax, or financial advice. Protect your signing keys: the examples use throwaway test keystores only.

Android, Google Play, the Google Play logo, Android Studio, Android vitals, and Google Cloud are trademarks of Google LLC. Gradle is a trademark of Gradle, Inc. Kotlin is a trademark of the Kotlin Foundation. GitHub and GitHub Actions are trademarks of GitHub, Inc. Java is a trademark of Oracle and/or its affiliates. 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, Gradle, Inc., JetBrains, or GitHub, Inc.

Northbeam Labs, the Waypoint app, Waypoint Plus, the Waypoint API (`api.waypoint.example`), and all people, trails, purchases, testers, and data in the running example are fictional. Email addresses and hostnames use reserved example domains. No real Google Play app, developer account, or payment profile was used.

Preface

Writing an Android app is only half the job. The other half starts when someone asks, "Can we ship it?" Suddenly the questions change. Which build goes to testers, and which one goes to the store? Why does the release build crash when the debug build was fine? Where is the signing key, who has the password, and what happens if the laptop that holds it dies? How do we roll out to one percent of users first, and how do we stop if something goes wrong? How do subscriptions actually reach our server, and how do we know a purchase is real?

Many teams answer these questions by accident. A build script grows one copied snippet at a time, the release is produced on one person's machine, a keystore sits in a shared folder, and the Play Console is clicked through from memory. It works — until the day it does not, usually on a Friday afternoon with users waiting.

This book replaces accidents with a deliberate, repeatable release process. We start from Gradle itself and work all the way to a staged production rollout of a paid subscription. Every chapter adds one piece to the same project, explains *why* the piece exists, and shows how to check that it works. By the end you will have a build you understand, a pipeline you trust, and a release routine you can hand to a teammate.

Who This Book Is For

- **Android developers** who can build an app in Kotlin and now need to own the build, the release, and the Play Console.
- **Team leads and senior developers** who want a clean multi-module build, fast CI, and a release process that does not depend on one person.
- **Indie developers** preparing a first serious launch, including subscriptions, who want to avoid the classic mistakes with keys, versions, and policies.
- **QA engineers and release managers** who work with testing tracks, staged rollouts, and crash data and want to understand what happens underneath.

You should be comfortable reading Kotlin and know what an Activity, a ViewModel, and a Compose screen are. You do not need prior Gradle plugin, CI, or Play Console experience.

What You Will Learn

- How Gradle and the Android Gradle Plugin really work: projects, tasks, the Kotlin DSL, dependencies, version catalogs, convention plugins, and the configuration and build caches.
- How to model debug, staging, and release builds and mock and production backends with build types, product flavors, BuildConfig fields, resources, manifest placeholders, source sets, and the Variant API.

- How to shrink and protect release builds with R8, write keep rules, use mapping files, and build and inspect Android App Bundles with bundletool.
- How app signing works on Google Play — upload keys, Play App Signing, key rotation and recovery — and how to keep keystores and passwords out of your repository.
- How to build continuous integration and release pipelines with GitHub Actions: tests, lint, artifacts, secrets, signed bundles, and automated uploads.
- How to use the Play Console: store listing, testing tracks, policy and Data safety declarations, staged rollouts, pre-launch reports, and release notes.
- How to sell a subscription with the Play Billing Library, verify purchases on your server, prompt for updates and reviews, monitor crashes and vitals, and roll back safely.

How This Book Is Organized

Part I — Gradle Foundations (Chapters 1–5) explains the release journey and the build system: Gradle basics, dependencies and version catalogs, convention plugins, and build performance. **Part II — Build Variants** (Chapters 6–10) covers build types, product flavors, per-variant values and resources, source sets, and the Variant API. **Part III — Shrinking, Packaging, and Signing** (Chapters 11–15) covers R8, keep rules, App Bundles and bundletool, app signing, and versioning. **Part IV — Continuous Integration** (Chapters 16–19) builds quality gates, pull-request checks, a release pipeline, and automated uploads to Google Play. **Part V — Google Play Console** (Chapters 20–23) covers app setup and store listing, testing tracks, policy and Data safety, and production releases. **Part VI — Monetization and Live Operations** (Chapters 24–28) covers Play Billing subscriptions, server-side verification, in-app updates and reviews, crash reporting and Android vitals, and the final release checklist and rollback playbook. The appendices provide a cheat sheet, templates and reference material, a production release checklist, and a glossary.

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

The Running Example: Waypoint

Throughout the book we work on **Waypoint**, a hiking trip planner and tracker made 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, and keep a private journal. Version 1.4.2 is already on Google Play. Our job is to ship **Waypoint 1.5.0**, the release that introduces **Waypoint Plus**, an optional subscription that unlocks offline trail packs and unlimited trips. Along the way the team — Lena Ortiz (Android lead), Sam Okafor (backend), Priya Natarajan (product and design), and Theo Brandt (QA) — turns a release that used to be built by hand on one laptop into a reproducible, automated, and monitored process.

The companion project is a small multi-module Gradle build with a mock backend, so you can build every variant, run every test, shrink with R8, and sign a bundle with a throwaway key on your own machine. Steps that need a real Google Play account — creating the app, uploading to tracks, buying a subscription — are described carefully and shown as what you should expect to see.

Versions and Conventions

The examples were checked with **Gradle 9.8**, **Android Gradle Plugin 9.4** with its built-in Kotlin support, **Kotlin 2.4**, **JDK 21**, compile SDK 37, target SDK 36, and minimum SDK 26, which were current stable releases at the time of writing. Build tools move quickly: if a newer version renames a DSL property or task, the concept stays the same, and a Note points out the places that changed recently. Google Play Console screens, policies, deadlines, and fees change often, so the book describes them as they were at the time of writing and always points you to the current official documentation. Command examples use Windows PowerShell and also show the macOS/Linux form where it differs, for example `.\gradlew.bat` versus `./gradlew`.

> **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 break a release, leak a secret, or violate a policy.

Gradle tasks, file names, and code appear in `monospace`; long lines are wrapped to fit the page. Play Console paths are written in bold with arrows, such as **Test and release > Production**; their exact labels can differ in your console. Variants follow the pattern flavor plus build type — `mockDebug`, `prodDebug`, `prodStaging`, and `prodRelease` — introduced in Part II and summarized in Appendix B.

Contents

Preface

PART I — GRADLE FOUNDATIONS

1 From Source to Store: How an Android Release Really Works

2 Gradle for Android Developers: Projects, Tasks, and the Kotlin DSL

3 Dependencies and Version Catalogs

4 Convention Plugins and a Clean build-logic

5 Fast, Reliable Builds: Configuration Cache, Build Cache, and Performance

PART II — BUILD VARIANTS

6 Build Types: Debug, Staging, and Release

7 Product Flavors, Flavor Dimensions, and Variant Filtering

8 BuildConfig, Resources, and Manifests per Variant

9 Source Sets and Variant-Specific Code

10 The Variant API: Customizing the Build with androidComponents

PART III — SHRINKING, PACKAGING, AND SIGNING

11 R8 Fundamentals: Shrinking, Obfuscation, and Optimization

12 Keep Rules, Mapping Files, and Debugging R8 Problems

13 Android App Bundles, APK Splits, and bundletool

14 App Signing: Upload Keys, Play App Signing, and Keystore Hygiene

15 Versioning Releases: versionCode, versionName, and Release Branches

PART IV — CONTINUOUS INTEGRATION

16 Quality Gates in the Build: Lint, Unit Tests, and Checks

17 GitHub Actions for Android: Build, Test, and Lint on Every Pull Request

18 Release Pipelines: Secrets, Signed Bundles, and Build Artifacts

19 Publishing from CI: Automated Uploads to Google Play

PART V — GOOGLE PLAY CONSOLE

20 Play Console Essentials: Developer Account, App Setup, and Store Listing

21 Testing Tracks: Internal, Closed, and Open Testing

22 Policy, Data Safety, and App Content Declarations

23 Production Releases: Staged Rollouts, Pre-Launch Reports, and Release Notes

PART VI — MONETIZATION AND LIVE OPERATIONS

24 Play Billing Fundamentals: Subscriptions, Base Plans, and Offers

25 Selling Waypoint Plus: Purchase Flow, Acknowledgement, and Server-Side Verification

26 In-App Updates and In-App Reviews

27 Crash Reporting, Monitoring, and Android Vitals

28 Shipping Safely: Release Checklist, Rollback, and the Waypoint 1.5.0 Launch

APPENDICES

A Build and Release Cheat Sheet

B Templates and Reference

C Production Release Checklist

D Glossary

PART I

Gradle Foundations

How an Android app travels from source code to a user's phone, and the build system that carries it there: Gradle projects and tasks in the Kotlin DSL, dependencies and version catalogs, convention plugins that keep a multi-module build tidy, and the caches and settings that make builds fast and repeatable.

1 · From Source to Store

2 · Gradle for Android Developers

3 · Dependencies and Version Catalogs

4 · Convention Plugins and build-logic

5 · Fast, Reliable Builds

CHAPTER 1 · PART I — GRADLE FOUNDATIONS

From Source to Store: How an Android Release Really Works

Between the moment you press Run in Android Studio and the moment a stranger opens your app on a phone in another country, a long chain of tools, files, keys, and decisions takes place. This chapter walks the whole chain once, introduces the team that will ship Waypoint 1.5.0 with us, and ends with your first build of the companion project.

Why This Book Exists

Most Android developers learn to build apps long before they learn to ship them. You can write a polished Compose screen, wire up a ViewModel, and call a REST API without ever thinking about what Gradle does, why a release build behaves differently from a debug build, or what happens to your app after you upload it to Google Play. For a while that works. Android Studio hides most of the machinery, and the first release of a small app is often done by one person, on one laptop, by following a blog post.

Then the app grows. Someone asks why the release build crashes when the debug build works perfectly. The person who knew where the signing key lives goes on vacation. A bad release reaches thousands of users before anybody notices, and there is no button to undo it. At that point the build and the release become a core engineering skill.

This book is about that skill. We will treat the build, the continuous integration pipeline, the release process, and the Play Console as one system that you can understand, automate, and trust. We will not teach Kotlin or Jetpack Compose; we assume you can already build an app. Instead we focus on everything that happens around your code: Gradle and the Android Gradle Plugin, build variants, R8, App Bundles, signing, versioning, quality gates, GitHub Actions, testing tracks, staged rollouts, Play Billing, in-app updates, crash reporting, and the habits that keep a release safe.

Meet Northbeam Labs and Waypoint

Every example in this book follows one fictional app and one fictional team, so that each chapter builds on the previous ones instead of starting from an unrelated toy project. The app is **Waypoint**, a hiking trip planner and tracker made by a small product studio called **Northbeam Labs**. Its package name and applicationId is `com.northbeam.waypoint`. Users browse trails such as Cedar Loop or Falcon Crest Trail, plan trips, record hikes with location samples, and keep a private trail journal.

Four people at Northbeam Labs will appear throughout the book. They are characters in the story, and each owns a different part of the release:

Person	Role	What they own in this book
Lena Ortiz	Android lead	The Gradle build, signing, CI, and every release
Sam Okafor	Backend	Purchase verification and real-time notifications
Priya Natarajan	Product and design	Store listing, release notes, paywall copy
Theo Brandt	QA	Testing tracks, pre-launch reports, release sign-off

If you are an indie developer, you wear all four hats; it helps to know which one you are wearing.

Where Waypoint stands today

When our story begins, **Waypoint 1.4.2** (versionCode `10402`) is live on Google Play. It works, users like it, and the reviews are good. But the way it got there is fragile. Lena built 1.4.2 by hand on her laptop. She selected the release variant in Android Studio, ran the signing wizard, typed the keystore password from memory, and uploaded the resulting file through the Play Console in a browser. The upload keystore lived on a USB stick in her desk drawer. Before the build she edited the versionCode by hand, and on one earlier release she forgot, so the upload was rejected and she had to rebuild late at night. There was no continuous integration: nobody ran the unit tests on every change, and lint warnings piled up unread.

Many successful apps ship exactly this way. The problem is that the process depends on one person remembering many steps, and no step leaves a record of which commit was built or who approved the upload.

The goal: Waypoint 1.5.0

The next release, **Waypoint 1.5.0** (versionCode `10500`), is a big one. It introduces **Waypoint Plus**, an optional subscription that unlocks offline trail packs and unlimited trips. Selling a subscription raises the stakes. A bug in the build can now cost money, a mistake in the store listing can delay review, and a crash in the purchase flow can generate refunds and one-star reviews. Lena decides that 1.5.0 will be the release where Northbeam Labs stops building by hand.

Over the next 28 chapters we will help her get there. We will clean up the Gradle build, add a staging build type for QA, make R8 safe, move the signing key into a vault and into CI secrets, derive the versionCode automatically, run tests and lint on every pull request, build a signed App Bundle from a Git tag, upload it to the internal testing track, set up the Play Console properly, sell Waypoint Plus with Play Billing, and finally ship 1.5.0 through a staged rollout. And because real releases do not always go to plan, Chapter 28 shows what happens when a crash appears at 5% of users, how Lena halts the rollout, and how the team ships hotfix **1.5.1** (versionCode `10501`).

The Journey from Source to Store

Before we look at any configuration file, it helps to see the whole path once, end to end. Each arrow in the following diagram is a topic that later chapters cover in depth.

The release pipeline at a glance

```
Kotlin + resources + manifests          (your code, in modules)
           |
           v
Gradle + Android Gradle Plugin          (Chapters 2-10: tasks, variants)
           |  compile, merge, dex, R8 (shrink + obfuscate)
           v
APK (installable)  or  AAB (publishing format)     (Chapters 11-13)
           |
           v
Sign with the UPLOAD key                (Chapter 14)
           |
           v
CI builds, tests, and uploads           (Chapters 16-19)
           |
           v
Google Play: re-sign with the APP SIGNING key,
generate device-specific APKs           (Chapters 13, 14, 20)
           |
           v
Tracks: internal -> closed -> (open) -> production   (Chapters 21-23)
           |
           v
Staged rollout: 1% -> 5% -> 20% -> 50% -> 100%       (Chapters 23, 28)
           |
           v
Users' devices -> crashes, ANRs, ratings, purchases
           |
           v
Android vitals, crash reports, reviews  (Chapters 26-27) -> next release
```

Step 1: Source code and resources

Everything starts with what you already know: Kotlin source files, XML or Compose UI, string resources, drawables, and an `AndroidManifest.xml`, spread over modules. Waypoint has six: the app plus five libraries. Some files exist in several versions; the launcher name is "Waypoint" in release but "Waypoint Debug" in debug. How Gradle picks the right one is the subject of Part II.

Step 2: Gradle and the Android Gradle Plugin

Gradle is a general-purpose build tool. On its own it knows nothing about Android. The **Android Gradle Plugin** (AGP) teaches it how to compile Kotlin for Android, merge manifests and resources from every module and library, convert JVM bytecode to the DEX format that Android runs, and

package the result. AGP also defines the concepts you meet every day in Android Studio: build types, product flavors, variants, source sets, and signing configurations. At the time of writing, Waypoint uses Gradle 9.8.0 and AGP 9.4.1 with built-in Kotlin support, which we explore in Chapter 2.

Step 3: Shrinking and optimization with R8

For release builds, AGP runs **R8**. R8 removes code and resources your app never uses, renames classes and members to short meaningless names, and optimizes the remaining code. On our machine, Waypoint's debug APK for the `prod` flavor is about 13.8 MB, while the shrunk release APK is about 1.56 MB. That difference is mostly unused library code that a debug build keeps to stay fast to build and easy to debug. R8 is also the most common reason why "it works in debug" but crashes in release, which is why Chapters 11 and 12 are devoted to it.

Step 4: APK or App Bundle

The build produces one of two formats. An **APK** (Android Package) is a file a device can install directly. An **Android App Bundle** (AAB) is a publishing format: it contains all the code and resources for every device configuration, and Google Play generates optimized APKs from it for each device. New apps on Google Play must be published as App Bundles; you still build APKs for testing (Chapter 13).

Step 5: Signing

Android refuses to install an app that is not signed, and it refuses to install an update that is signed by a different key than the installed version. Signing is therefore not optional, and losing control of a key is serious. With **Play App Signing**, there are two keys. You sign your bundle with an **upload key** that proves the upload came from you. Google Play verifies it, then re-signs the APKs it generates with the **app signing key**, which Google keeps on your behalf. Debug builds are signed with a debug key that Android Studio creates automatically on every developer machine. Chapter 14 covers all of this in detail.

Step 6: Continuous integration

Continuous integration (CI) runs the same Gradle commands on a clean machine for every pull request, and a release pipeline builds the signed bundle from a Git tag. Waypoint uses GitHub Actions, but the ideas transfer to any CI system (Part IV).

Step 7: Google Play tracks

An uploaded bundle goes to a **track**, not to all users. Internal testing reaches a small list of testers quickly, closed testing reaches invited groups, open testing lets anyone join, and production reaches everyone you choose. Moving a tested release from one track to the next is called promotion (Part V).

Step 8: Staged rollout and monitoring

A **staged rollout** makes a production release available to a percentage of users first. Northbeam's own rule for 1.5.0 is 1%, then 5%, 20%, 50%, and finally 100%, with at least 24 hours per step and a set of gates that must pass before the next step. If something goes wrong, the rollout can be halted. What you cannot do is take an update back from devices that already installed it: the only way forward is a new release with a higher versionCode (Chapters 23 and 28).

Step 9: Users, vitals, and the next release

Once users have the release, crashes and ANRs ("Application Not Responding" errors) flow into Android vitals and your crash-reporting SDK, and ratings and subscription events arrive. These signals decide whether a rollout continues and what goes into the next release (Chapter 27).

> **NOTE**
>
> Google Play's requirements, policies, quotas, and console screens change frequently. Everything in this book about the Play Console describes it at the time of writing. When a number, deadline, or menu path matters for your release, check the current Play Console Help, the Policy Center, and the Android Developers documentation. Labels and locations change; search the console if a menu has moved.

Debug Builds and Release Builds

The most important distinction in the whole pipeline is the one between a debug build and a release build. They come from the same source code, but they are different programs in practice. Many release problems come from forgetting how different they are.

Aspect	Debug build	Release build
Purpose	Fast iteration on your machine	What users install
Debuggable	Yes: debugger, layout inspector	No
R8 shrinking	Off (in Waypoint)	On: code and resources
Signing key	Debug key, created automatically	Upload key, kept secret
Size (Waypoint, our machine)	About 13.8 MB APK	About 1.56 MB APK
Build speed	Fast, incremental	Slower: R8 and lint checks

A debug build keeps every class with its original name and can be debugged. A release build is shrunk and obfuscated by R8, checked by the critical `lintVital` lint checks, and signed with a key only your team controls. Testing only debug builds therefore tells you little about what users run, so Waypoint adds a third build type, **staging**: a copy of release with its own applicationId and the debug signing key, which QA installs next to the Play version (Chapter 6).

> **WARNING**
>
> Never assume a release build behaves like a debug build. Reflection, JSON serialization of classes R8 renamed, resources loaded by name, and libraries without proper keep rules can all work in debug and fail in release. Test a minified build, such as Waypoint's `prodStaging` variant, before every release, not after users report crashes.

A Tour of the Companion Project

This book comes with a companion project: the Waypoint source code in the shape it has at the end of the book. It builds green with the toolchain listed in the next section. You do not need to type it in; we read it, change copies of it, and explain why each part exists. When a chapter needs an experiment that would break the project, we do it in a scratch copy so that the original stays clean.

Companion project layout (abridged)

```
sample/
  settings.gradle.kts        project name "waypoint", modules, build-logic
  build.gradle.kts           root: declares plugins once with apply false
  gradle.properties          build settings + waypoint.versionName=1.5.0
  gradle/libs.versions.toml  the version catalog (Chapter 3)
  gradle/wrapper/            Gradle 9.8.0 wrapper with checksum
  keystore.properties.example   template for local release signing
  build-logic/               convention plugins (Chapter 4)
  app/                       the Android application module
  core/model                 pure Kotlin: Trail, Trip, User, SampleData
  core/network               Retrofit API for the fictional backend
  core/data                  TrailRepository + a fake implementation
  core/billing               Waypoint Plus: Play Billing + a fake
  feature/trails             trail list screen (Compose) and ViewModel
  tools/new-test-keystore.ps1   creates a throwaway signing key
  distribution/whatsnew/     release notes for Google Play
  .github/workflows/         ci.yml and release.yml
```

Modules and their direction

The dependency direction is strict and points one way: the app depends on everything, the trails feature depends on the data layer, the data layer depends on networking, and networking depends on the models. The `core:model` module is plain Kotlin with no Android dependency at all, which makes its tests fast. The billing module stands on its own and is used only by the app.

```
app --> feature:trails --> core:data --> core:network --> core:model
 |
 +--> core:billing, core:data, core:network, core:model
```

Four variants

Waypoint has two **product flavors** in one flavor dimension called `backend`: `mock`, which uses fake data and a fake billing implementation so the app runs without a server or a Play account, and `prod`, which talks to the real (fictional) backend and to Google Play. It has three **build types**: `debug`, `staging`, and `release`. Two flavors times three build types would give six variants, but the mock flavor never ships, so the build disables every mock variant except debug. That leaves exactly four:

Variant	applicationId	versionName	Launcher name
mockDebug	com.northbeam.waypoint .mock.debug	1.5.0-debug	Waypoint Mock
prodDebug	com.northbeam.waypoint .debug	1.5.0-debug	Waypoint Debug
prodStaging	com.northbeam.waypoint .staging	1.5.0-staging	Waypoint Staging
prodRelease	com.northbeam.waypoint	1.5.0	Waypoint

(The applicationIds are split with a space in this table only to fit the page; the real values have no space.) All four variants share versionCode `10500`. Because their applicationIds differ, a tester can install all four on one phone side by side. Only `prodRelease` is ever uploaded to Google Play. Part II explains how each value is produced.

Workflows and tools

The `.github/workflows` folder holds two GitHub Actions workflows. `ci.yml` has one job called `build` that runs on every pull request and on pushes to `main`: unit tests, lint, debug builds, and an unsigned release bundle. `release.yml` has one job called `bundle` that runs when a tag such as `v1.5.0` is pushed; it builds a bundle signed with the upload key and uploads it to the internal testing track. We write both in Part IV. The `tools/new-test-keystore.ps1` script creates a throwaway signing key for experiments, so that you never need a real key to follow along.

The toolchain

The companion project was built and verified with the following versions, which were the newest stable releases at the time of writing:

Tool	Version
Gradle (wrapper)	9.8.0
Android Gradle Plugin	9.4.1 (built-in Kotlin)
Kotlin	2.4.20
JDK for Gradle	21 (bytecode target 17)
compileSdk / targetSdk / minSdk	37 / 36 / 26
Compose BOM	2026.09.00

Play Billing Library (billing-ktx)	9.1.0

Newer versions sometimes rename a DSL property or change a default; when a behavior depends on a version, we say so.

Your First Build

Let us build Waypoint. You need JDK 21 (Android Studio ships one) and the Android SDK with platform 37. You do not install Gradle: the project's **Gradle wrapper** downloads the exact declared Gradle version and runs it. Every command in this book goes through the wrapper.

Work on a copy

Keep the companion project as a clean reference and experiment in a copy. Several chapters break something on purpose, and a fresh copy is also exactly what a CI machine sees.

When you copy the project, leave out the folders that Gradle and the IDE generate: `build`, `.gradle`, `.kotlin`, and `.idea`. Also leave out any signing material and `local.properties`, which holds the path to your Android SDK and is specific to each machine. On Windows, `robocopy` can do this in one command:

PowerShell

```
robocopy C:\books\waypoint C:\work\waypoint-ch01 /E `
  /XD build .gradle .kotlin .idea `
  /XF local.properties keystore.properties *.jks *.keystore
```

On macOS or Linux, `rsync -a --exclude build --exclude .gradle ...` does the same job, or you can simply clone the repository again with Git. Android Studio recreates `local.properties` when you open the project; from the command line, create it yourself with one line that points at your SDK:

local.properties (example for Windows)

```
sdk.dir=C\:\\Users\\you\\AppData\\Local\\Android\\Sdk
```

The path differs on every machine, so never commit this file; Waypoint's `.gitignore` lists it.

Run the build

Open a terminal in the project folder and build the `mockDebug` variant. It is the friendliest variant to start with, because it needs no server, no Play account, and no signing key.

PowerShell (Windows)

```
.\gradlew.bat assembleMockDebug
```

macOS / Linux / CI

```
./gradlew assembleMockDebug
```

In PowerShell, call the wrapper with the `.\` prefix. Some Windows shell configurations refuse to run a program from the current folder without it, and the prefix makes your intent explicit either way. The first build of a fresh copy takes a while: the wrapper may download Gradle, Gradle downloads dependencies, the daemon starts, and every module compiles. On our machine, a shared workstation with warm dependency and build caches, the first build of a fresh copy took 48 seconds in the run shown below and 18 seconds in a quieter run (Chapter 5); on a new machine with empty caches, expect a few minutes.

Output (abridged)

```
> Task :app:preBuild UP-TO-DATE
> Task :core:network:preDebugBuild UP-TO-DATE
> Task :app:generateMockDebugBuildInfo
...
> Task :feature:trails:compileDebugKotlin FROM-CACHE
> Task :app:compileMockDebugKotlin FROM-CACHE
> Task :app:dexBuilderMockDebug FROM-CACHE
> Task :app:mergeProjectDexMockDebug FROM-CACHE
> Task :app:packageMockDebug
> Task :app:assembleMockDebug
> Task :app:createMockDebugApkListingFileRedirect

BUILD SUCCESSFUL in 48s
118 actionable tasks: 51 executed, 67 from cache
Configuration cache entry stored.
```

Reading the output

That output looks like noise the first time, but it tells a precise story. Each line that starts with `> Task` is one unit of work. The part before the last colon is the project path, so `:feature:trails:compileDebugKotlin` is the Kotlin compilation of the `feature:trails` module, and `:app:packageMockDebug` is the packaging step of the app module for the `mockDebug` variant. Notice that the library modules compile their `Debug` variant: they have no flavors, so the app's `mockDebug` variant consumes each library's `debug` variant.

The word after a task name is its **outcome**. No word means the task ran. `UP-TO-DATE` means Gradle checked the task's inputs and outputs and found nothing to do. `FROM-CACHE` means Gradle found the outputs in the build cache, a store of earlier results keyed by the task inputs, and copied them instead of running the task. You will also see `NO-SOURCE` (the task had no input files, for example a Java compilation in a module with only Kotlin) and `SKIPPED`. Our first build shows many `FROM-CACHE` outcomes because the build cache is enabled in `gradle.properties` and our machine had built the same code before. On your machine the first build will run most tasks.

The summary line, "118 actionable tasks: 51 executed, 67 from cache", counts the tasks that could do work. The last line, "Configuration cache entry stored.", means Gradle saved the result of the configuration phase, so the next identical command can skip it. Chapter 2 explains the phases and Chapter 5 explains both caches in depth.

Now run the same command again without changing anything:

Output (abridged)

```
Reusing configuration cache.
> Task :app:generateMockDebugAssets UP-TO-DATE
> Task :app:preBuild UP-TO-DATE
...
> Task :app:packageMockDebug UP-TO-DATE
> Task :app:assembleMockDebug UP-TO-DATE

BUILD SUCCESSFUL in 4s
114 actionable tasks: 114 up-to-date
Configuration cache entry reused.
```

On our machine, four seconds instead of 48: every task is `UP-TO-DATE` and the configuration cache entry was reused. In a healthy build, work happens only when something changed. If a no-op build of your project still runs tasks, some task is not declaring its inputs and outputs correctly.

> **TIP**
>
> Add `--console=plain` when you want to copy build output into a bug report. The default rich console redraws progress lines in place, which produces messy captured text.

Finding the APK

The APK lands in a folder that follows a fixed pattern, `app/build/outputs/apk/` followed by the flavor and the build type:

Build outputs

```
app/build/outputs/apk/mock/debug/app-mock-debug.apk       (about 13.8 MB)
app/build/outputs/apk/mock/debug/output-metadata.json
```

Next to the APK, AGP writes `output-metadata.json`, a small machine-readable description of what it produced. Scripts and CI jobs can read it instead of guessing file names:

app/build/outputs/apk/mock/debug/output-metadata.json

```
{
  "version": 3,
  "artifactType": {
    "type": "APK",
    "kind": "Directory"
  },
  "applicationId": "com.northbeam.waypoint.mock.debug",
```

```
  "variantName": "mockDebug",
  "elements": [
    {
      "type": "SINGLE",
      "filters": [],
      "attributes": [],
      "versionCode": 10500,
      "versionName": "1.5.0-debug",
      "outputFile": "app-mock-debug.apk"
    }
  ],
  "elementType": "File",
  "minSdkVersionForDexing": 26
}
```

Look at the values. The applicationId has both suffixes, `.mock` from the flavor and `.debug` from the build type. The versionName carries the `-debug` suffix. The versionCode is `10500`, even though nobody typed that number anywhere: the build derives it from `waypoint.versionName=1.5.0` in `gradle.properties`. Lena will never bump a versionCode by hand again. Chapter 15 shows the rule, and Chapter 4 shows where the code that applies it lives.

Waypoint also writes its own small report for every variant. A custom Gradle task generates `build-info.json` and AGP packages it into the APK as an asset, so a support engineer, or the app itself, can tell exactly which variant is running:

app/build/generated/assets/generateMockDebugBuildInfo/build-info.json

```
{
  "variant": "mockDebug",
  "applicationId": "com.northbeam.waypoint.mock.debug",
  "versionName": "1.5.0-debug",
  "versionCode": 10500
}
```

We will write that task in Chapter 10 with the Variant API. For now, notice that you just saw its name in the build output: `:app:generateMockDebugBuildInfo`.

Installing on a device

If you have an emulator running or a phone connected with USB debugging enabled, you can install the variant directly. AGP creates an install task for every variant that is signed:

PowerShell

```
.\gradlew.bat installMockDebug
```

You should see something like this at the end of the output: