51 · Deployment

Flutter Practice Notes

Contents

51 · Deployment

Deployment Fundamentals

Store Setup & Submission

Review Process & Compliance

App Size & Build Optimization

Deployment Integration (Capstone: Full Release + Post-Launch)

51 · Deployment

Introduction

This module covers **shipping a Flutter app to production**: the **fundamentals** (release channels, store overview, versioning), **store setup & submission** (Play Console / App Store Connect — listings, metadata, assets, build upload), the **review process & compliance** (how review works, common rejections, privacy/policy requirements), and **app-size & build optimization** (App Bundle/thinning, obfuscation, shrinking the download), tied together in a **full release + post-launch** capstone. It's the last mile after CI/CD (Module 50) and the handoff to monitoring (Module 52).

Why this module exists

Getting an app *into users' hands* is a distinct discipline from building it: the stores have their own **setup, metadata, assets, review, and policy** requirements, releases must be **versioned and staged** correctly, rejected submissions cost days, and a bloated download hurts installs/conversion. Knowing the **store processes, compliance rules, and size-optimization levers** — plus post-launch practices — is what turns a finished build into a live, discoverable, compliant, well-performing product.

Module map

#	File	Topic	Level
1	01_deployment_fundamentals.md	Release channels, store overview, versioning, release strategy	🔵
2	02_store_setup_and_submission.md	Play Console / App Store Connect setup, listing, metadata, assets, upload	🟡
3	03_review_process_and_compliance.md	Review process, common rejections, privacy/policy compliance	🔴
4	04_app_size_and_build_optimization.md	App Bundle/thinning, obfuscation, size reduction, release build	🔴
5	05_deployment_integration.md	Capstone: full release checklist + post-launch	🟡

> **Cross-references:** CI/CD (automates build/upload): Module 50. Monitoring (post-launch): Module 52. Signing/flavors: 50 · build_signing. App size/startup: Module 21. Security/privacy: Module 37. Native config: Module 27/Module 28. Payments policy: Module 31.

Prerequisites

50 CI CD (signing/build/release automation), native build config (27/28), 21 Performance (app size).

What you'll be able to do after this module

- Explain release channels, store distribution, and versioning strategy.
- Set up Play Console / App Store Connect listings with correct metadata + assets and upload builds.
- Navigate the review process, avoid common rejections, and meet privacy/policy requirements.
- Reduce app size (App Bundle/thinning/obfuscation) and produce optimized release builds.
- Run a full release (checklist + staged rollout) and handle post-launch.

Capstone

Release + post-launch: Take an app to production — configure the store listing (metadata, screenshots, privacy details, content rating), upload a signed optimized App Bundle/IPA, submit for review with a compliance checklist, do a staged rollout, and set up post-launch practices (monitoring, responding to reviews, hotfix process) — as a documented release runbook.

Summary

Deployment is the last mile: version and channel your release, set up compliant store listings with metadata/assets, pass review (avoiding common rejections + meeting privacy/policy), ship an optimized (App Bundle/thinned/obfuscated) build via staged rollout, and run post-launch practices. It turns a finished, CI/CD-built app into a live, discoverable, compliant product.

Deployment Fundamentals

> Deployment is getting your app to users through the **app stores** — **Google Play** (Android) and the **Apple App Store** (iOS) — which mediate distribution via **release channels/tracks** (internal → beta → production), **review**, and **updates**. The two anchors: **artifact format** — Android ships an **Android App Bundle (`.aab`)** (Google generates optimized per-device APKs via **App Bundle + dynamic delivery**), iOS ships an `.ipa` to App Store Connect — and **versioning** — a user-facing **semantic version** (`1.2.0`) plus a **monotonically increasing build number** (stores **reject duplicates**). Plan a **release strategy** (cadence, channels, staged rollout) rather than shipping ad hoc.

Introduction

This file establishes the deployment landscape: the stores, channels/tracks, artifact formats, versioning, and a release-strategy mindset — the frame for the setup/review/optimization files. It's the "how apps reach users" foundation.

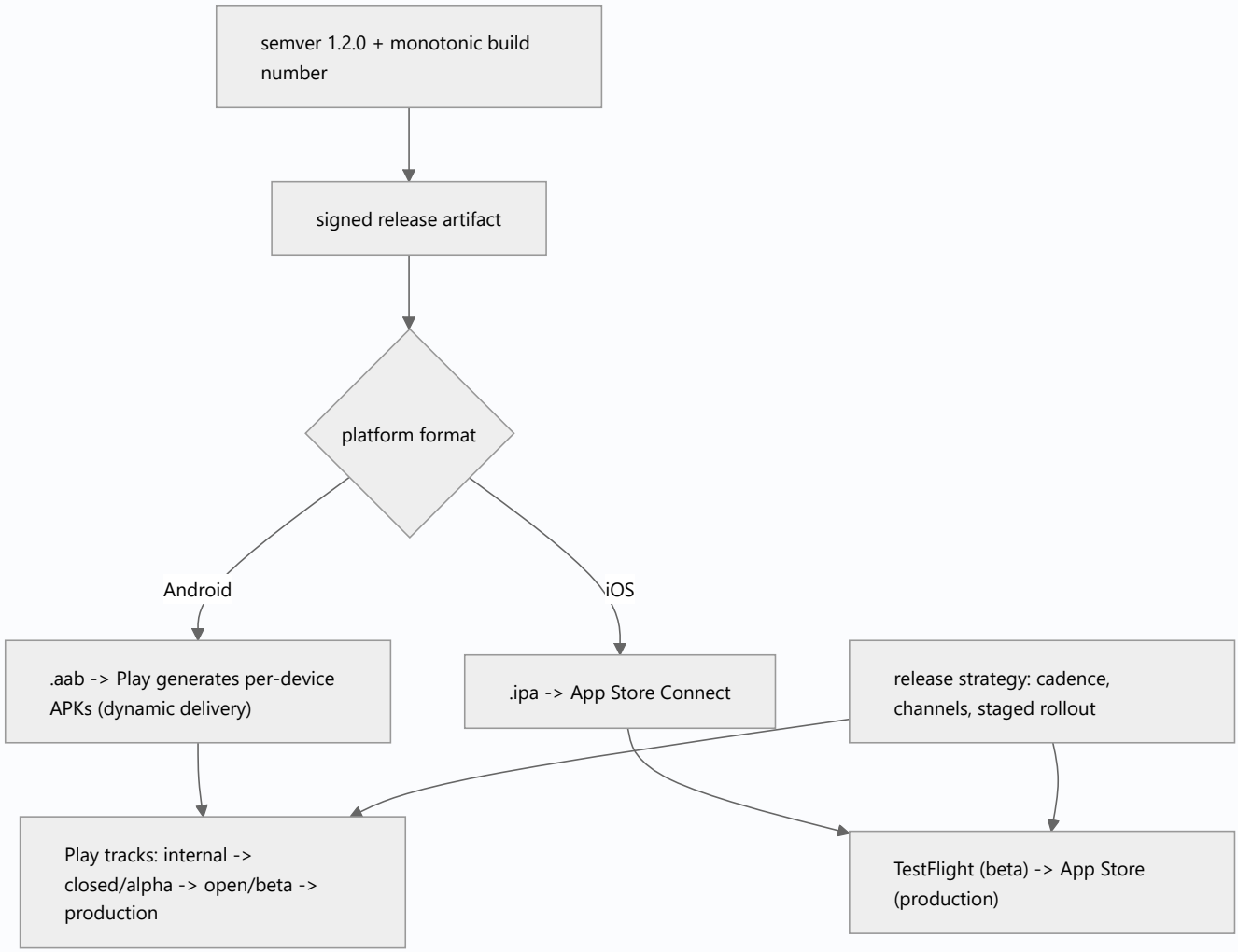
Why this concept exists

Unlike web (deploy to your server), mobile distribution is **gatekept by stores** with their own formats, review, versioning rules, and update mechanics. Understanding this — App Bundle vs APK, version vs build number, tracks, review latency — prevents the classic failures (rejected uploads, duplicate build numbers, wrong channel) and lets you plan releases deliberately.

Real-world analogy

The stores are **regulated retailers you must go through** to reach shelves: you can't hand products directly to customers — you submit to the retailer (store), pass **inspection** (review), and stock **back-room test shelves** (internal/beta) before the **main floor** (production). Each product batch needs a unique **lot number** (build number) and a **version label** (semver). You plan **restock cadence + phased shelf placement** (release strategy/staged rollout), not random dumping.

Internal Working



- **The stores**:
  - **Google Play** (Android): Play Console; faster/automated review (hours-days); staged rollout %; multiple tracks.
  - **Apple App Store** (iOS): App Store Connect; human review (typically ~1 day, variable); TestFlight for beta; Phased Release for production.
  - (Others exist — Amazon Appstore, Huawei AppGallery, enterprise/MDM, web/desktop stores — but Play + App Store dominate.)
- **Artifact formats (know these)**:
  - **Android — Android App Bundle (`.aab`)**: the **required** publishing format. You upload one `.aab`; **Google Play generates + serves optimized APKs per device** (dynamic delivery: only the code/resources/density/ABI that device needs) → **smaller downloads**. (Raw `.apk` is for sideload/testing, not Play publishing.) `flutter build appbundle`.
  - **iOS — `.ipa`**: uploaded to App Store Connect (via Xcode/Transporter/Fastlane); Apple handles device optimization (app thinning/slicing). `flutter build ipa`.

- **Release channels / tracks (progressive exposure)** (Module 50):
  - **Play**: **internal** (small team, instant) → **closed/alpha** → **open/beta** → **production** (with **staged rollout %**).
  - **App Store**: **TestFlight** (internal + external testers) → **App Store** (production, Phased Release).
  - Promote up as confidence grows; never straight to 100% production.
- **Versioning (get it right or uploads fail)**:
  - `version: 1.2.0+34` in `pubspec.yaml` → **version name** (`1.2.0`, user-facing **semver**) + **build number** (`34`, Android `versionCode` / iOS `CFBundleVersion`).
  - The **build number must strictly increase** per upload — stores **reject duplicates/lower**. Automate the bump (Module 50).
  - **Semver**: MAJOR.MINOR.PATCH communicates change scope; bump per release.
- **Release strategy (plan, don't wing it)**: decide **cadence** (weekly/biweekly/on-demand), **channel flow** (internal→beta→prod), **staged rollout** (5%→100% + monitoring — Module 52), and **who approves production** (gated — Module 50). Mobile releases are **hard to roll back** (users must update) → **forward-fix + staged rollout**.
- **Accounts/enrollment (prereqs)**: **Google Play Developer** account (one-time fee) + **Apple Developer Program** (annual fee); Apple requires a **Mac + Xcode** for building/uploading iOS.
- **Automate via CI/CD**: build/sign/upload to tracks are automated (Module 50); this module is the store-side knowledge that automation targets.

Memory Representation

Not runtime — a **release plan + artifacts**: the `.aab` / `.ipa`, version+build-number metadata, track/channel state, and a cadence/rollout strategy. Stores hold the track state + review status.

Compiler / Build Behavior

`flutter build appbundle` / `ipa` produce signed release artifacts (AOT-compiled, per flavor); version/build number embed at build; App Bundle enables Play's per-device APK generation.

Runtime Behavior

Users download the store-optimized artifact (smaller via App Bundle/thinning); production rolls out progressively (staged/phased); updates arrive via store auto-update. No instant rollback.

Flutter Engine Behavior

Release builds ship AOT-compiled Dart + the engine; App Bundle strips unneeded ABIs/resources per device (04_app_size_and_build_optimization.md).

Dart VM Behavior

Release = AOT (no JIT/VM overhead in prod); tree-shaking/obfuscation reduce size (Module 21).

Examples

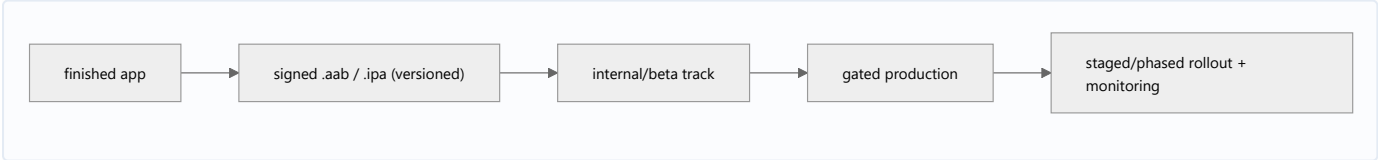
```
# pubspec.yaml — version name + build number (build number MUST increase per upload)
version: 1.2.0+34    # 1.2.0 = semver (user-facing), 34 = build number (versionCode/CFBundleVersion)
```

```
Build artifacts:
  flutter build appbundle --release --flavor prod   # Android -> .aab (Play publishing format)
  flutter build ipa       --release --flavor prod   # iOS     -> .ipa (App Store Connect)
  # (flutter build apk = sideload/test only, NOT for Play publishing)

Channel/track flow (never straight to 100% prod):
  Play:      internal -> closed/alpha -> open/beta -> production (staged 5%->100%)
  App Store: TestFlight (internal/external) -> App Store (Phased Release)
```

```
Release strategy checklist:
  cadence (e.g., biweekly) | channel flow (internal->beta->prod) | staged rollout % + monitoring
  | gated prod approval | auto build-number bump | forward-fix plan (mobile rollback is hard)
```

Diagrams



Common Mistakes

Mistake	Why it's a problem	Fix
Uploading a raw APK to Play	Not the publishing format	Build/upload an App Bundle (`.aab`)
Duplicate/lower build number	Store rejects upload	Auto-increment a unique monotonic build number
Straight to 100% production	Bad build hits everyone	Staged/phased rollout + monitoring
No release strategy (ad hoc)	Chaotic, risky releases	Plan cadence/channels/rollout/approval
Expecting easy rollback	Users already have the build	Forward-fix + staged rollout + flags
Confusing version vs build number	Rejected/mislabeled releases	semver = user-facing; build number = unique/increasing
No Mac for iOS	Can't build/upload iOS	Mac + Xcode (or Mac CI runner)

Best Practices

- Ship the right **artifact**: Android **App Bundle (`.aab`)** (Play generates per-device APKs → smaller downloads); iOS `.ipa` to App Store Connect.
- Version correctly: **semver version name** + a **strictly increasing build number** (`1.2.0+34`); **automate the build-number bump**; never reuse/lower it.
- Use **channels/tracks** (internal → beta → production) and **staged/phased rollout** with **monitoring**; **gate production**; plan **cadence + approval** (a real release strategy).
- Design for **forward-fix** (mobile rollback is hard); automate build/sign/upload via **CI/CD** (Module 50); ensure prereqs (developer accounts, Mac for iOS).

Performance

Deployment affects install/runtime performance: **App Bundle/thinning** shrink the download (higher install conversion), **AOT release builds + obfuscation/tree-shaking** reduce size + startup (04_app_size_and_build_optimization.md/Module 21). Staged rollout limits the blast radius of a bad release.

Advantages / Disadvantages

- **+** Reaches users via trusted stores, per-device-optimized downloads (App Bundle/thinning), staged safe rollout, structured channels.
- **−** Store gatekeeping (review latency/policy), hard rollback, versioning strictness, account/Mac prerequisites, per-store processes.

Interview Questions

1. 🟢 **What artifact do you publish on each platform?** — Android: an Android App Bundle (`.aab`) — Play generates optimized per-device APKs; iOS: an `.ipa` to App Store Connect.
2. 🟢 **Version name vs build number?** — Version name = user-facing semver (`1.2.0`); build number = a strictly increasing integer (`versionCode` / `CFBundleVersion`) — stores reject duplicates.
3. 🟡 **Why an App Bundle instead of an APK for Play?** — Play uses it for dynamic delivery — serving only the code/resources/density/ABI each device needs → smaller downloads; raw APK is for sideload/testing.
4. 🟡 **What are release channels/tracks, and why progressive?** — Play internal→beta→production, App Store TestFlight→App Store; promote up as confidence grows and roll out staged, so a bad build reaches few.
5. 🟡 **Why can't you easily roll back a mobile release?** — Users already have the installed build; you mitigate with staged rollout + halt, feature flags, and forward-fix (hotfix).
6. 🔴 **What does a release strategy include?** — Cadence, channel flow, staged rollout % + monitoring, gated production approval, auto build-number bump, and a forward-fix plan.
7. 🔴 **What are the prerequisites to publish?** — Google Play Developer + Apple Developer Program accounts, and a Mac + Xcode (or Mac CI runner) for iOS builds/uploads.

Senior Engineer Tips

- Always publish an App Bundle and auto-increment the build number in CI; "uploaded an APK" and "duplicate versionCode" are the two most common first-release failures.
- Never go straight to 100% production — internal→beta→staged rollout with monitoring is the standard, and it's your main defense given mobile's hard rollback.
- Write a release runbook (cadence, channels, rollout, approval, forward-fix) early; ad hoc releases are where avoidable incidents happen.

Architect Perspective

Deployment fundamentals frame mobile's store-gatekept distribution: the right artifact (App Bundle/ `.ipa`), correct versioning (semver + monotonic build number), progressive channels, and a deliberate staged, gated release strategy built around forward-fix (since rollback is hard). Getting these right — and automating them via CI/CD — turns a built app into a reliably shippable product and sets up the store-setup, review, and optimization work that follows (02_store_setup_and_submission.md, Module 50, Module 52).

Summary

- Publish via stores: Android App Bundle (`.aab` , per-device APKs) / iOS `.ipa` ; version = semver name + strictly increasing build number (`1.2.0+34`).
- Use channels (internal→beta→production) + staged/phased rollout + monitoring; gate production; plan a real release strategy (cadence/approval).
- Mobile rollback is hard → forward-fix + staged rollout + flags; automate build/sign/upload via CI/CD; need dev accounts + Mac for iOS.

Revision Notes

- Stores: Google Play (`.aab` , dynamic delivery → per-device APKs; faster review; staged rollout %) / Apple App Store (`.ipa` , App Store Connect, TestFlight beta, Phased Release; human review; needs Mac/Xcode).
- Version: `pubspec version: 1.2.0+34` → name (semver, user-facing) + build number (versionCode/CFBundleVersion, strictly increasing — no duplicates); auto-bump.
- Channels: Play internal→closed/alpha→open/beta→production; App Store TestFlight→App Store; staged/phased rollout + monitoring; gated prod. Rollback hard → forward-fix/flags. Prereqs: Play Developer + Apple Developer accounts, Mac for iOS. Automate via CI/CD.

Practice Questions

1. Why publish an App Bundle rather than an APK on Play?
2. What's the difference between version name and build number, and why does it matter?
3. What does a good release strategy include?

Coding Questions

1. Set `pubspec` version + build number correctly and build both platform artifacts.
2. Outline the channel/track flow from internal to production for both stores.
3. Draft a release-strategy checklist (cadence/channels/rollout/approval/forward-fix).

Mini Project

Release plan (Flutter/deployment): For an app, define the deployment plan: artifact formats (`.aab` / `.ipa`), versioning scheme (semver name + auto-incrementing build number), channel/track flow (internal→beta→staged production for both stores), staged-rollout + monitoring + gated-approval strategy, and a forward-fix rollback plan — plus prerequisites (accounts, Mac).

Acceptance: correct artifacts + versioning (unique/increasing build number); progressive channel flow (not straight-to-prod); staged rollout + monitoring + gated approval; forward-fix plan; prerequisites listed; CI/CD-automatable.

Store Setup & Submission

> Publishing requires a fully configured **store listing** plus the **build upload**. In **Play Console** and **App Store Connect** you create the app, fill **metadata** (name, description, keywords, category), provide **visual assets** (icon, screenshots per device size, feature graphic/preview), complete **compliance forms** (privacy policy, **data-safety/App Privacy** disclosures, **content rating**, target audience), set up **pricing/availability**, and upload the **signed artifact** to a **track**. Getting the **assets sized correctly**, the **privacy disclosures accurate**, and the **build on the right track** is what turns a submission into a live (or reviewable) app — and what avoids instant rejections.

Introduction

This file is the practical checklist for both consoles: account/app creation, metadata, assets (with sizing), the compliance/privacy forms, pricing/availability, and uploading a build to a track. It's the store-side execution of the fundamentals (01_deployment_fundamentals.md).

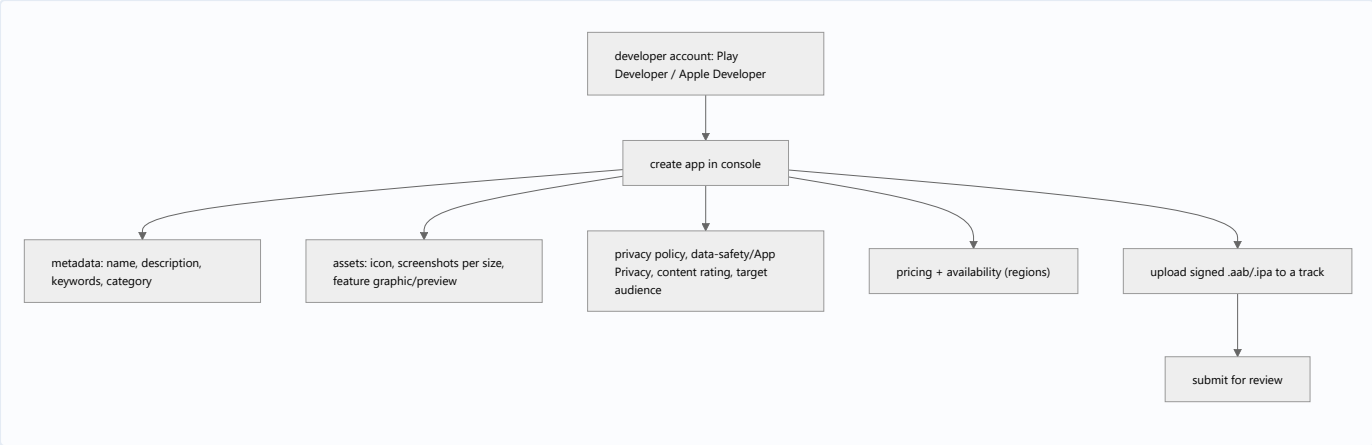
Why this concept exists

A build alone isn't publishable — stores require a complete, compliant **listing** (how users discover/evaluate the app) and mandatory **disclosures** (privacy/rating). Missing or wrong assets/metadata/privacy answers cause rejections or block submission. Knowing exactly what each console needs (and correct asset specs) makes the first submission smooth instead of a multi-day back-and-forth.

Real-world analogy

It's **setting up a retail product listing**: you register as a seller (developer account), create the **product page** (metadata + photos), fill mandatory **regulatory labels** (privacy/ingredients/age rating), set **price + regions**, and deliver the **inventory** (the build) to the right **shelf** (track). A page with the wrong-size photos, missing labels, or product on the wrong shelf won't go live.

Internal Working



- **Prereqs**: **Google Play Developer** account (one-time fee) + **Apple Developer Program** (annual). Create the app in **Play Console** / **App Store Connect** (set bundle/application id — must match your signed build + flavor).
- **Metadata** (discovery + evaluation):
  - **App name/title**, **short + full description** (Play) / **subtitle + description** (App Store), **keywords** (App Store field; Play uses description for ASO), **category**, **contact info**, **support/marketing URL**, **promotional text**.
  - Optimize for **ASO** (app store optimization): clear title, keyword-relevant description, compelling first screenshots.
- **Visual assets (size them exactly — wrong sizes are rejected)**:
  - **App icon** (adaptive on Android; specific px on iOS).
  - **Screenshots** per required **device size** (phone + tablet/iPad; iOS requires specific display sizes) — the primary conversion driver; show real value in the first 1–2.
  - **Play**: **feature graphic** (1024×500) + optional promo video (YouTube).
  - **App Store**: optional **app preview** video; **localized** assets per language.
  - Provide **localized** metadata/assets for target markets.
- **Compliance/privacy forms (mandatory — block submission if missing/wrong)** (03_review_process_and_compliance.md/Module 37):
  - **Privacy policy URL** (required if you collect any data).
  - **Data safety (Play)** / **App Privacy "nutrition labels" (App Store)**: **accurately** declare what data you collect, why, whether it's shared/linked to identity, and tracking (**ATT** on iOS). **Must match actual app behavior** — mismatches cause rejection/removal.
  - **Content rating** (Play questionnaire → IARC rating) / **age rating** (App Store); **target audience/kids** declarations (extra rules if targeting children — COPPA/Families).
  - **Permissions justification** (sensitive permissions may need declaration forms, esp. Android — background location, etc.).

- **Pricing & availability**: free/paid, in-app purchases config, **country/region availability**, and (paid apps) tax/banking setup.
- **Build upload → track**:
  - **Play**: upload the `.aab` to a **track** (internal/closed/open/production); fill **release notes**; set **staged rollout %** for production.
  - **App Store**: upload the `.ipa` (Xcode/Transporter/Fastlane `pilot` / `deliver`) → appears in **TestFlight** / select for an **App Store version**; attach screenshots + "what's new".
  - Ensure the **build's version/build number** and **bundle id** match the console + are unique/increasing (01_deployment_fundamentals.md).
- **Submit for review**: once listing + build + compliance are complete, **submit**; track review status. Automate metadata/asset/build upload with **Fastlane** (`supply` / `deliver`) / Codemagic (Module 50).
- **First-submission gotchas**: missing screenshots for a required device size, inaccurate data-safety answers, no privacy policy URL, mismatched bundle id, undeclared sensitive permissions — all block/reject.

Memory Representation

Not runtime — a **console configuration**: app record + metadata + localized assets + compliance answers + pricing + uploaded build on a track. Fastlane can store metadata/assets as **files in the repo** (`fastlane/metadata`, `screenshots/`) for version-controlled, automated submission.

Compiler / Build Behavior

The uploaded artifact must be a **signed release** build with matching **bundle id + version/build number**; console validates format/signing/version on upload (rejects duplicates/unsigned/wrong-id).

Runtime Behavior

Not applicable (store-side). Once approved + rolled out, users see the listing + download the build.

Flutter Engine Behavior

Not applicable.

Dart VM Behavior

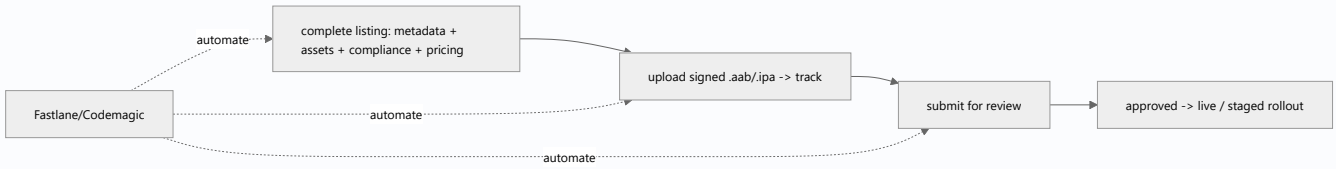
Not applicable.

Examples

```
Submission checklist (both stores):
  [ ] Developer account (Play Developer / Apple Developer) + app created (bundle id matches build)
  [ ] Metadata: name, description(s), keywords (iOS), category, support URL, promo text
  [ ] Assets: icon; screenshots for EVERY required device size; Play feature graphic (1024x500); (optional preview video)
  [ ] Compliance: privacy policy URL; Data Safety (Play) / App Privacy (App Store) ACCURATE; content/age rating; target audience
  [ ] Permissions: justify/declare sensitive ones (e.g., background location)
  [ ] Pricing & availability (regions, IAP config, tax/banking if paid)
  [ ] Upload signed .aab/.ipa to a track; release notes; (prod) staged rollout %
  [ ] Localize metadata/assets for target markets
  [ ] Submit for review
```

```
# Fastlane — version-controlled metadata/assets + automated upload
# deliver   # (App Store) uploads screenshots + metadata from fastlane/metadata, submits
# supply(track: 'internal', aab: 'build/app.aab', metadata_path: 'fastlane/metadata/android')
```

Diagrams



Common Mistakes

Mistake	Why it blocks/rejects	Fix
Missing screenshots for a required device size	Submission blocked	Provide all required sizes (phone + tablet/iPad)
Inaccurate data-safety/App Privacy answers	Rejection/removal	Declare data usage to match actual behavior
No privacy policy URL (but collecting data)	Blocked/rejected	Provide a valid privacy policy URL
Bundle id mismatch (build vs console)	Upload rejected	Match application/bundle id + flavor
Duplicate/lower build number	Upload rejected	Unique increasing build number
Undeclared sensitive permissions	Rejection (esp. Android)	Complete permission declaration forms
Unsigned/debug build uploaded	Rejected	Upload signed release artifact
No localized assets for target markets	Poor discovery	Localize metadata/screenshots

Best Practices

- Complete the **full listing** (accurate metadata + correctly-sized, compelling **screenshots per device**, feature graphic/preview) and **localize** for target markets; optimize for **ASO** (first screenshots + keyword-relevant description).
- Fill **compliance forms accurately** (privacy policy URL, **Data Safety/App Privacy matching real behavior**, content/age rating, target audience, sensitive-permission declarations) — mismatches cause rejection/removal.
- Upload a **signed release** `.aab` / `.ipa` with **matching bundle id + unique increasing build number** to the correct **track**, with release notes (+ staged rollout % for prod).
- **Automate** metadata/asset/build upload with **Fastlane (** `supply` / `deliver` **)**/Codemagic; keep metadata/assets **version-controlled**.

Performance

Not runtime perf — but **listing quality drives install conversion** (screenshots/description = ASO). App size (from the build) affects install rates (04_app_size_and_build_optimization.md). Automating submission speeds release cadence.

Advantages / Disadvantages

- **+** Discoverable, compliant, professional listing; automatable + version-controlled (Fastlane); localized reach; correct track distribution.
- **−** Many exact requirements (asset sizes, forms) to satisfy; per-store differences; accurate-privacy discipline; first-time setup is fiddly.

Interview Questions

1. 🟢 **What's needed to publish beyond the build?** — A complete store listing: metadata, correctly-sized assets (icon/screenshots/feature graphic), compliance forms (privacy/data-safety/rating), pricing/availability — then upload the signed build to a track and submit.
2. 🟢 **What visual assets do the stores require?** — App icon and screenshots for each required device size (phone + tablet/iPad); Play also needs a 1024×500 feature graphic; optional preview videos; localized per market.
3. 🟡 **What are Data Safety / App Privacy forms, and why critical?** — Mandatory disclosures of what data you collect/share/track; they must match actual app behavior or the app is rejected/removed.
4. 🟡 **What common setup mismatches block an upload?** — Bundle-id mismatch, duplicate/lower build number, unsigned/debug build, missing required screenshots, no privacy policy URL.

5. 🟡 **How do you handle sensitive permissions in submission?** — Complete permission declaration/justification forms (esp. Android background location) and disclose in privacy forms.
6. 🔴 **How do you automate/version-control submission?** — Fastlane `supply` (Play) / `deliver` (App Store) upload metadata/assets/build from repo files; integrate into CI/CD.
7. 🔴 **Why does listing quality matter (ASO)?** — Title, description, and especially the first screenshots drive discovery + install conversion; localize for target markets.

Senior Engineer Tips

- Prepare and version-control your metadata/screenshots (Fastlane `metadata` / `screenshots` dirs) so submissions are automated and repeatable — hand-entering the console every release is slow and error-prone.
- Fill privacy/data-safety forms to match what the app actually does, and keep them updated when you add data collection; inaccurate disclosures are a rejection/removal risk, not a formality.
- Have all required screenshot sizes + a valid privacy policy URL + matching bundle id/build number ready before your first submission; those are the usual multi-day blockers.

Architect Perspective

Store setup/submission is the productization layer: a compliant, discoverable listing + a correctly-uploaded signed build is what makes an app publishable. Treating metadata/assets/compliance as **version-controlled, automated** artifacts (Fastlane in CI/CD) — with accurate privacy disclosures and correct sizing/versioning — turns submission from a fragile manual ritual into a repeatable pipeline step, feeding the review process and staged rollout (03_review_process_and_compliance.md, Module 50).

Summary

- Configure a full listing (metadata + correctly-sized/localized assets + accurate compliance forms + pricing) and upload a signed `.aab` / `.ipa` (matching bundle id + unique build number) to a track, then submit.
- Compliance (privacy policy, Data Safety/App Privacy, content rating, permission declarations) is mandatory + must match real behavior.
- Optimize the listing for ASO; automate + version-control submission with Fastlane/Codemagic.

Revision Notes

- Prereqs: Play Developer + Apple Developer accounts; create app (bundle id matches build/flavor). Metadata: name/description/keywords(iOS)/category/URLs/promo (ASO).
- Assets (exact sizes, localized): icon; screenshots per required device size (phone + tablet/iPad); Play feature graphic 1024×500; optional preview video.
- Compliance (mandatory, match behavior): privacy policy URL; Data Safety (Play)/App Privacy (App Store); content/age rating; target audience/kids; sensitive-permission declarations. Pricing/availability + IAP + tax/banking.
- Upload signed `.aab` / `.ipa` (matching bundle id + unique increasing build number) → track + release notes (+ staged rollout %); submit. Automate via Fastlane `supply` / `deliver`, version-controlled metadata/assets.

Practice Questions

1. What must a complete store listing include beyond the build?
2. Why must Data Safety/App Privacy forms match actual behavior?
3. What common mismatches block a build upload?

Coding Questions

1. Draft a submission checklist for both stores (metadata/assets/compliance/upload).
2. Structure version-controlled Fastlane metadata/screenshots for automated upload.
3. Identify + fix three setup errors that would block a first submission.

Mini Project

Store submission setup (Flutter/deployment): Prepare a full submission for both stores: a metadata set (name/description/keywords/category), an asset spec (icon + required screenshot sizes + Play feature graphic, localized), accurate compliance answers (privacy policy URL, Data Safety/App Privacy, content rating, permission declarations), pricing/availability, and the signed-artifact upload plan (matching bundle id + unique build number → track + release notes + staged rollout) — automated/version-controlled via Fastlane. Acceptance: complete listing (metadata + correctly-sized/localized assets); accurate compliance forms (matching real behavior); signed artifact with matching id + unique build number → correct track; ASO-optimized; automatable/version-controlled submission; no common blockers.

Review Process & Compliance

> Both stores **review** submissions against their **policies** before (and after) publishing — **Apple** does thorough **human review** (~a day, stricter, subjective), **Google Play** is faster/more automated (hours–days) but enforces policy just as firmly (with post-publish scanning). The frequent **rejection causes** are predictable: **crashes/bugs/incomplete apps**, **inaccurate privacy/data-safety disclosures**, **missing/incorrect permissions justification**, **payment-policy violations** (using a gateway for digital goods instead of IAP), **guideline breaches** (misleading metadata, prohibited content, broken links), and **incomplete listings**. Compliance — **privacy, permissions, IAP, content, and platform guidelines** — is a first-class release requirement, not an afterthought.

Introduction

This file covers how review works on each store, the most common rejection reasons and how to avoid them, and the key compliance areas (privacy, permissions, payments, content, guidelines). It's the "getting approved + staying compliant" layer after submission (02_store_setup_and_submission.md).

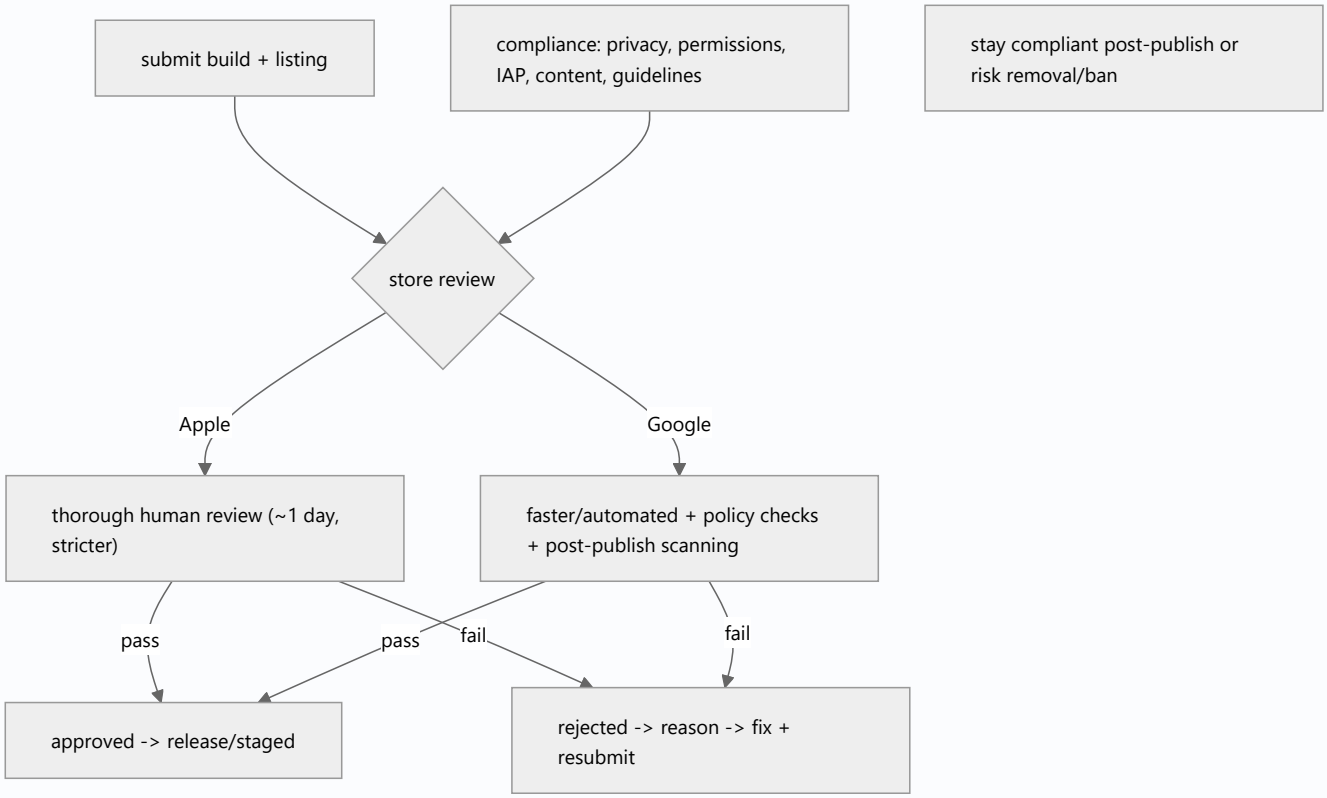
Why this concept exists

Stores gatekeep quality, safety, privacy, and their business rules; a rejection costs days and delays launches, and post-publish violations can get an app **removed** or an account **banned**. Knowing the process + common pitfalls + compliance rules lets you submit **right the first time** and avoid costly, sometimes reputation-damaging, rejections/removals.

Real-world analogy

Review is **customs + safety inspection** before your product hits shelves: inspectors check it **works, is safe, is truthfully labeled** (privacy/permissions), and follows **trade rules** (payment policy). Apple's inspection is a **thorough manual one** (slower, stricter, some judgment); Google's is **faster/more automated** but still enforced, with **random re-inspections** (post-publish scanning). A mislabeled, buggy, or rule-breaking product is turned away — repeatedly if you don't fix the root cause.

Internal Working



- **The review process**:
  - **Apple App Store**: **human reviewers** test the app against the **App Store Review Guidelines**; typically **~24h** (variable); **stricter + more subjective** (design/quality/completeness judged). Rejections come with a reason (in App Store Connect **Resolution Center**); you fix + resubmit or appeal.
  - **Google Play**: **faster, largely automated** review (hours to a few days) plus **policy enforcement + ongoing scanning** (malware/policy post-publish). Rejections/warnings via Play Console.
  - **Pre-launch reports / TestFlight**: Play's **pre-launch report** (auto-tests on devices) surfaces crashes before production; TestFlight beta catches issues with real testers.
- **Common rejection causes (know + prevent these)**:
  - **Crashes / bugs / incomplete features**: the #1 cause — test thoroughly (unit/widget/E2E — Module 49), handle errors, no placeholder/broken UI, no dead links.
  - **Inaccurate privacy disclosures**: Data Safety/App Privacy **not matching** actual data collection (02_store_setup_and_submission.md/Module 37).
  - **Permissions**: requesting permissions **without clear justification/usage strings**, or more than the app needs (Module 27/Module 28); Android sensitive-permission declarations.
  - **Payment-policy violations**: using an **external gateway for digital goods** (must use **IAP**) — a top Apple/Google rejection (Module 31).

    - **Guideline breaches**: **misleading metadata/screenshots**, prohibited/objectionable content, **spam/duplicate** apps, **misuse of trademarks**, insufficient **account-deletion** support (now required), inadequate **content moderation** for user-generated content.
    - **Design/quality (Apple)**: too-basic "website wrapper" apps, poor UX, non-native feel.
    - **Incomplete listing / demo access**: missing screenshots, or **no test credentials** for a login-gated app (provide a demo account for reviewers).
- **Compliance areas (first-class)**:
    - **Privacy**: privacy policy, accurate disclosures, **ATT** (iOS tracking prompt), **account deletion** path, data-minimization (Module 37).
    - **Permissions**: request **only what's needed**, with **clear justification/usage strings**, in context (Module 29).
    - **Payments**: **IAP for digital goods**, gateways for physical/services (Module 31).
    - **Content**: rating accuracy, no prohibited content, moderation + reporting for UGC.
    - **Legal/regional**: age/kids rules (COPPA/Families), GDPR/CCPA, export/encryption declarations, accessibility expectations.
- **Handling rejections**: read the **exact reason**, fix the **root cause** (not just symptoms), respond in the **Resolution Center**/appeal with clarification if it's a misunderstanding, and **resubmit**. Repeated same-cause rejections signal you didn't fix the root.
- **Staying compliant post-publish**: policies **evolve**; violations found later → **removal/suspension/ban**. Keep disclosures current, respond to policy notices, and monitor (Module 52).
- **Reduce risk**: use **pre-launch reports/TestFlight**, submit a **complete, crash-free, honestly-disclosed** app, and **budget review time** (esp. Apple) into release schedules.

Memory Representation

Not runtime — a **compliance state + review status**: policy adherence (privacy/permissions/IAP/content), submission status, rejection reasons/history. The invariant: the app + listing truthfully comply with current policies.

Compiler / Build Behavior

Review evaluates the built/signed artifact + listing; some checks (e.g., export-compliance/encryption declarations, entitlements) tie to build config; a crashing build fails review.

Runtime Behavior

Reviewers/automated tools **run the app** (Apple manual, Play pre-launch report) — so runtime **crashes/broken flows** are caught. Post-publish scanning continues at runtime scale.

Flutter Engine Behavior

Not applicable beyond the app running normally under review.

Dart VM Behavior

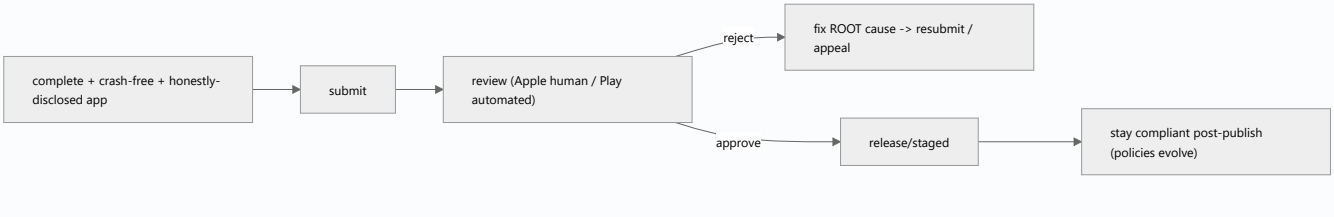
Not applicable.

Examples

```
Top rejection causes -> preventions:
  crashes/incomplete/broken links  -> thorough testing (unit/widget/E2E), no placeholders, working links
  inaccurate privacy disclosures   -> Data Safety/App Privacy MATCH actual behavior
  unjustified/excess permissions   -> request only needed + clear usage strings/justification
  gateway for DIGITAL goods         -> use IAP (Module 31)
  misleading metadata/screenshots  -> honest, representative listing
  login-gated app, no demo creds   -> provide reviewer test account + notes
  UGC without moderation/reporting -> add moderation + report/block + account deletion
```

```
Review reality:
  Apple:  human review, ~24h (variable), stricter/subjective (quality/design/completeness)
  Google: faster/automated (hours-days) + policy scanning post-publish
  -> budget review time (esp. Apple); use pre-launch report / TestFlight to catch issues first
```

Diagrams



Common Mistakes

Mistake	Why it's rejected/risky	Fix
Crashes/incomplete/placeholder UI	#1 rejection cause	Test thoroughly; no broken flows/links
Inaccurate privacy/data-safety	Rejection/removal	Disclosures match real behavior
Excess/unjustified permissions	Rejection	Request only needed + clear justification/usage strings
Gateway for digital goods	Payment-policy violation	Use IAP (Module 31)
Misleading metadata/screenshots	Guideline breach	Honest, representative listing
No reviewer demo credentials (login app)	Reviewer can't test → reject	Provide test account + review notes
UGC without moderation/reporting/deletion	Policy breach	Add moderation, reporting, account deletion
Ignoring root cause on resubmit	Repeated rejections	Fix root cause, respond in Resolution Center
No review-time budget	Missed launch dates	Budget review latency (esp. Apple)

Best Practices

- Submit a **complete, crash-free** app (thorough testing, working links, no placeholders) and **honest, accurate disclosures** (privacy/data-safety matching behavior, correct ratings).
- Request **only necessary permissions** with **clear justification/usage strings**; use **IAP for digital goods**; keep **metadata/screenshots truthful**; support **account deletion** + **UGC moderation/reporting** where applicable.
- Provide **reviewer demo credentials + notes** for gated apps; use **pre-launch reports/TestFlight** to catch issues first; **budget review time** (esp. Apple).
- On rejection, **fix the root cause** + respond/appeal in the Resolution Center; **stay compliant post-publish** (policies evolve) and monitor.

Performance

Not runtime perf — but **crash-free + performant** apps pass review (crashes are the top rejection) and avoid post-publish removal, so quality/monitoring feed compliance (Module 49/Module 52). Review latency affects **release cadence** (budget it).

Advantages / Disadvantages

- **+** (Compliance) smooth first-pass approval, avoided removals/bans, user trust, predictable release schedule.

- **−** Review latency (esp. Apple) + subjectivity, evolving policies to track, per-store rules, rejection back-and-forth if unprepared.

Interview Questions

1. 🟢 **How do Apple and Google review differ?** — Apple: thorough human review (~a day, stricter/subjective on quality/design); Google: faster/more automated (hours-days) with post-publish policy scanning.
2. 🟢 **What's the #1 rejection cause and how to avoid it?** — Crashes/bugs/incomplete apps — prevent with thorough testing (unit/widget/E2E), no placeholders/broken links, and pre-launch reports/TestFlight.
3. 🟡 **What compliance areas matter most?** — Privacy (accurate disclosures, ATT, account deletion), permissions (only needed + justified), payments (IAP for digital goods), content (rating/moderation), and platform guidelines.
4. 🟡 **Why do inaccurate privacy disclosures get apps removed?** — Data Safety/App Privacy must match actual data behavior; mismatches are a policy violation → rejection or post-publish removal.
5. 🟡 **How do you help reviewers test a login-gated app?** — Provide demo/test credentials and review notes so the reviewer can access the full app.
6. 🔴 **What happens if you use a payment gateway for digital goods?** — Rejection — digital goods must use IAP; gateways are only for physical goods/services.
7. 🔴 **How do you handle a rejection?** — Read the exact reason, fix the root cause (not symptoms), respond/appeal in the Resolution Center if it's a misunderstanding, and resubmit — repeated same-cause rejections mean the root wasn't fixed.

Senior Engineer Tips

- Ship complete and crash-free with honest disclosures; the vast majority of rejections are crashes, inaccurate privacy answers, or payment-policy violations — all preventable before submitting.
- Always include reviewer demo credentials + notes for gated apps and use Apple's + Google's pre-launch tooling; "reviewer couldn't log in" and "crashed on launch" are needless multi-day delays.
- Budget Apple review time into release schedules and fix the root cause on rejection (not the symptom); and keep privacy/policy disclosures current post-launch, since policies evolve and violations get apps pulled.

Architect Perspective

Review and compliance are the gate between a built app and a live one: quality (crash-free/tested), truthful privacy/permissions, correct payments (IAP), and guideline adherence. Treating compliance as a first-class release requirement — baked into testing, privacy disclosures, permission design, and payment architecture — turns review into a formality rather than a blocker, and prevents the costlier post-publish removals. It ties the whole handbook's quality/security/payments work to the moment of shipping (Module 49, Module 37, Module 31).

Summary

- Stores review submissions against policy (Apple: human/stricter ~1 day; Google: faster/automated + post-publish scanning); rejections cost days.
- Top rejections: crashes/incomplete, inaccurate privacy disclosures, unjustified permissions, gateway-for-digital-goods, misleading metadata, no reviewer demo creds, UGC without moderation.
- Compliance (privacy/permissions/IAP/content/guidelines) is first-class; submit complete + honest, provide demo access, use pre-launch tools, fix root causes, stay compliant post-publish.

Revision Notes

- Review: Apple human (~24h, stricter/subjective, Resolution Center) vs Google faster/automated (hours-days) + ongoing policy scanning; pre-launch report (Play)/TestFlight catch issues first.
- Top rejections: crashes/incomplete/broken links; inaccurate privacy/data-safety; unjustified/excess permissions; gateway-for-digital-goods (use IAP); misleading metadata; no reviewer demo credentials (login apps); UGC without moderation/reporting/account-deletion.
- Compliance: privacy (accurate disclosures + ATT + account deletion), permissions (only needed + justification/usage strings), payments (IAP digital), content (rating/moderation), legal/regional. Fix root cause on rejection; budget review time (Apple); stay compliant post-publish (removal/ban risk).

Practice Questions

1. What are the most common rejection causes and their preventions?
2. How does Apple review differ from Google Play review?
3. What compliance areas must you get right, and what happens if you don't?

Coding Questions

1. Build a pre-submission compliance checklist (privacy/permissions/IAP/content/quality).
2. Draft reviewer notes + demo-credential handling for a login-gated app.
3. Given a rejection reason, outline the root-cause fix + resubmission response.

Mini Project

Compliance + review readiness (Flutter/deployment): For an app, produce a review-readiness package: a compliance checklist (crash-free/tested, accurate privacy disclosures, minimal justified permissions, IAP-for-digital-goods, honest metadata, account deletion, UGC moderation if applicable), reviewer demo credentials + notes for a gated flow, a pre-launch-testing plan (Play pre-launch report/TestFlight), and a rejection-handling process (root-cause fix + Resolution Center response) with review-time budgeting. Acceptance: covers top rejection causes + preventions; accurate privacy/permissions/payments/content compliance; reviewer demo access; pre-launch testing; root-cause rejection handling + review-time budget; post-publish compliance noted.

App Size & Build Optimization

> Download size directly affects **install conversion** (bigger = fewer installs, esp. on limited data/storage), so shrink it: publish an **Android App Bundle** (Play serves per-device APKs via **dynamic delivery** — only the ABI/density/resources that device needs) and let iOS **app-thin/slice**; build `--release` (AOT, tree-shaken) with `--obfuscate --split-debug-info` (smaller + harder to reverse — keep the mapping to de-symbolicate crashes); **compress/right-size assets** (images, fonts — subset), **remove unused deps/assets**, and **defer** heavy features (deferred loading). **Measure** with `--analyze-size` and the App Bundle explorer, and set an **app-size budget** you enforce.

Introduction

This file covers reducing app size + producing optimized release builds: App Bundle/thinning, release-build flags (AOT/obfuscation/tree-shaking), asset optimization, dependency hygiene, deferred loading, and measurement/budgets. It applies performance techniques (Module 21) to the deployment artifact.

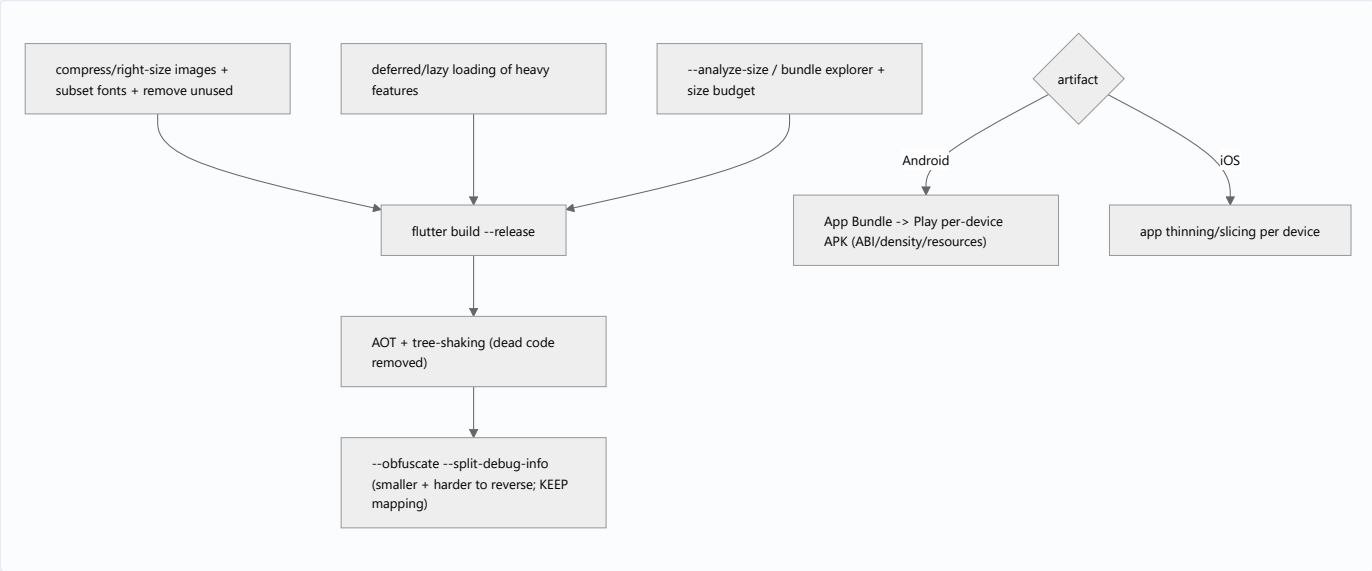
Why this concept exists

Every megabyte costs installs — users on limited data/storage abandon large downloads, and stores surface size. A default release build can be much larger than necessary (all ABIs, uncompressed assets, unused code). Knowing the size levers (App Bundle, thinning, obfuscation, assets, deferral) turns a bloated artifact into a lean one, improving conversion + startup without cutting features.

Real-world analogy

Shipping an app is like **mailing a package**: you don't send the whole warehouse — you **pack only what this recipient needs** (App Bundle/thinning = per-device contents), **use compact packaging** (compressed/right-sized assets), **remove filler** (unused deps/code via tree-shaking), and **ship bulky extras separately, on request** (deferred loading). A lean package arrives faster and more people accept the delivery (install conversion).

Internal Working



- **Publish an App Bundle (Android) / rely on thinning (iOS)** (01_deployment_fundamentals.md):
  - `flutter build appbundle` → Play's **dynamic delivery** serves each device only its **ABI** (arm64 vs armv7 vs x86), **screen density**, and **language resources** → the on-device download is **much smaller** than a universal APK.
  - iOS: App Store **app thinning/slicing** delivers per-device variants automatically.
  - (If you must ship APKs, `--split-per-abi` avoids a fat universal APK.)
- **Release build flags**:
  - `--release`: **AOT-compiled**, **tree-shaken** (unused Dart code + unused icon-font glyphs removed) — never ship debug.
  - `--obfuscate --split-debug-info=<dir>`: renames symbols (smaller + reverse-engineering cost) and writes the **debug-info mapping** — **archive it** to **de-symbolicate crash reports** (Module 37/Module 52).
- **Asset optimization (often the biggest win)**:
  - **Images**: compress (WebP/optimized PNG/JPEG), provide **appropriate resolutions** (avoid shipping 4K for a thumbnail), use **vector**/ `flutter_svg` where suitable, and prefer **network/CDN + cached** images over bundling large media (Module 34).
  - **Fonts**: **subset** fonts to used glyphs (`--tree-shake-icons` handles icon fonts); avoid bundling many full font families.
  - **Remove unused assets** from `pubspec.yaml`.
- **Dependency hygiene**: **remove unused packages** (each adds code/assets); prefer lean deps; be wary of packages pulling large native libs. Audit with size analysis.
- **Deferred / lazy loading**: split rarely-used/heavy features so their code loads **on demand** rather than in the base download — Flutter **web** supports deferred imports; on mobile, **Play**

feature delivery / dynamic feature modules can defer install-time modules (advanced). Keep the **initial** download lean.

- **Native/platform**: enable Android **resource shrinking/R8** (Gradle `shrinkResources` / `minifyEnabled` for the Android layer), strip debug symbols, and remove unused native libs.
- **Measure + budget**:
  - `flutter build appbundle --analyze-size` (and `--analyze-size` for apk/ipa) + the **App Bundle Explorer** / **DevTools app-size tool** → see what's big (Dart/assets/native/fonts).
  - Set an **app-size budget** and enforce it in **CI** (Module 50/Module 47); investigate regressions as features land.
- **Startup benefit**: smaller + AOT + deferred also improves **startup/runtime** (Module 21) — size and startup optimization overlap.

Memory Representation

Not runtime state — a **build artifact composition**: Dart AOT code (tree-shaken/obfuscated), assets (compressed/subset), native libs (per-ABI via App Bundle), and (optionally) deferred modules loaded on demand. Size analysis reports this breakdown.

Compiler / Build Behavior

`--release` triggers AOT + tree-shaking; `--obfuscate` renames symbols (+ mapping); `--tree-shake-icons` drops unused glyphs; App Bundle enables Play's per-device splitting; R8/resource-shrinking trims the Android layer. Deterministic with pinned toolchains.

Runtime Behavior

Users download a **per-device-optimized, smaller** artifact (App Bundle/thinning) → faster install + less storage; deferred features load on demand; AOT gives fast startup. Obfuscated crash traces need the mapping to read.

Flutter Engine Behavior

The engine ships in the artifact; App Bundle/thinning include only the needed **ABI** per device; tree-shaking removes unused framework code paths where possible.

Dart VM Behavior

Release = **AOT** (no JIT/VM in prod → smaller/faster); tree-shaking removes dead Dart; obfuscation shrinks symbol names. Deferred imports defer code loading (web).

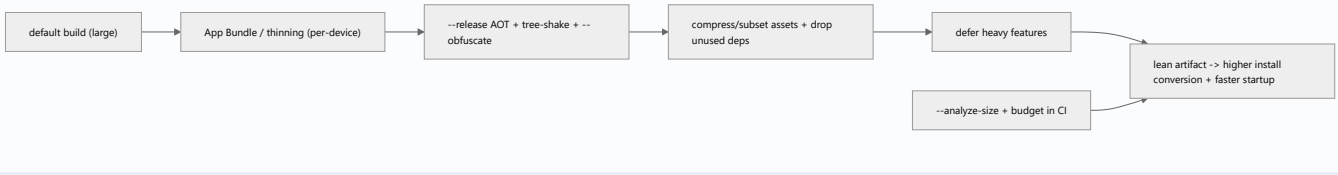
Examples

```
Build lean release artifacts:
  flutter build appbundle --release --obfuscate --split-debug-info=build/symbols   # Android (App Bundle)
  flutter build ipa       --release --obfuscate --split-debug-info=build/symbols   # iOS
  # archive build/symbols per release to de-symbolicate crashes (Module 52)

Measure size (find what's big):
  flutter build appbundle --analyze-size          # Dart/assets/native/fonts breakdown
  # + Play Console App Bundle Explorer (per-device download size), DevTools app-size tool
```

```
Size levers (biggest wins first, typically):
  App Bundle (per-device delivery)  ....... large win vs universal APK
  compress/right-size images + WebP ....... often the biggest asset win
  --release AOT + tree-shaking + --tree-shake-icons
  --obfuscate (smaller + security), keep mapping
  remove unused deps/assets
  R8/resource shrinking (Android)
  defer heavy/rare features (deferred loading)
  -> enforce an app-size BUDGET in CI
```

Diagrams



Common Mistakes

Mistake	Why it bloats/hurts	Fix
Shipping a universal APK	Includes all ABIs → huge	App Bundle (per-device) / `--split-per-abi`
Shipping a debug build	Large + slow (no AOT)	`--release`
Not obfuscating (or losing mapping)	Bigger + readable / unreadable crashes	`--obfuscate --split-debug-info`, archive mapping
Bundling huge/uncompressed images	Biggest bloat source	Compress/right-size/WebP; CDN + cache
Bundling full font families	Wasted glyphs	Subset fonts; `--tree-shake-icons`
Unused deps/assets left in	Dead weight	Remove; audit with size analysis
No size measurement/budget	Silent regression	`--analyze-size` + CI size budget
Everything in the base download	Large initial size	Defer heavy/rare features

Best Practices

- Publish an **App Bundle** (Android per-device delivery) / rely on **iOS thinning**; always build `--release` with `--obfuscate --split-debug-info` (archive the mapping for crash de-symbolication).
- **Optimize assets** (compress/right-size/WebP images, **subset fonts**, `--tree-shake-icons`, prefer CDN+cache for large media) and **remove unused deps/assets**; enable **R8/resource shrinking**.
- **Defer heavy/rare features** (deferred loading) to keep the **initial download lean**.
- **Measure** with `--analyze-size` + bundle explorer, set an **app-size budget**, and **enforce it in CI** (investigate regressions as features land).

Performance

Size optimization is a **conversion + startup** win: smaller downloads install more (esp. limited-data markets), and AOT + tree-shaking + deferral also cut **startup time/memory** (Module 21). App Bundle/thinning are the biggest structural wins; assets are usually the biggest content win. A CI size budget prevents gradual bloat.

Advantages / Disadvantages

- **+** Higher install conversion, faster startup, less storage/data, security (obfuscation), per-device efficiency, regression-guarded (budget).
- **−** Some setup (obfuscation mapping management, asset pipeline, deferred loading complexity), measurement discipline, deferred-module complexity on mobile.

Interview Questions

1. 🟢 **How does an App Bundle reduce download size?** — Play's dynamic delivery serves each device only its ABI, screen density, and language resources, versus a universal APK containing everything.
2. 🟢 **What release flags shrink the build?** — `--release` (AOT + tree-shaking, incl. `--tree-shake-icons`) and `--obfuscate --split-debug-info` (smaller symbols + mapping) — never ship debug.
3. 🟡 **What's usually the biggest content contributor to size, and how to reduce it?** — Images/assets — compress/right-size (WebP, appropriate resolutions), subset fonts, prefer CDN+cache for large media, remove unused assets.
4. 🟡 **Why keep the `--split-debug-info` mapping?** — Obfuscated release crash traces are unreadable without it; you need it to de-symbolicate production crashes.
5. 🟡 **How do you keep the initial download lean for a big app?** — Defer heavy/rare features (deferred loading / dynamic feature modules) so they load on demand, not in the base download.
6. 🔴 **How do you measure + control app size?** — `flutter build --analyze-size` + Play App Bundle Explorer/DevTools app-size tool to find big contributors; set an app-size budget enforced in CI.
7. 🔴 **How do size and startup optimization relate?** — They overlap: AOT + tree-shaking + smaller assets + deferral reduce both download size and startup time/memory.

Senior Engineer Tips

- Always publish an App Bundle and audit assets first — per-device delivery + image/font optimization typically deliver the largest size wins with the least effort.
- Turn on obfuscation and archive the symbol mapping per release; you get smaller + more secure builds and can still read production crashes.
- Set an app-size budget and check it in CI (`--analyze-size`); like performance, size rots gradually as features land unless a guardrail catches regressions.

Architect Perspective

App-size/build optimization is a conversion + performance concern realized at the artifact level: App Bundle/thinning (structural), release AOT + obfuscation + tree-shaking (build), asset/dependency hygiene (content), and deferral (loading) — measured against a CI-enforced budget. It's the deployment-side of performance discipline: a lean, per-device-optimized, obfuscated artifact installs better, starts faster, and stays that way as the app grows — feeding install conversion and post-launch crash de-symbolication (Module 21, Module 50, Module 52).

Summary

- Publish an App Bundle (Android per-device delivery) / iOS thinning; build `--release` with `--obfuscate --split-debug-info` (archive mapping).
- Optimize assets (compress/right-size/WebP, subset fonts, tree-shake icons, CDN+cache), remove unused deps/assets, R8/resource shrinking, defer heavy features.
- Measure (`--analyze-size` + bundle explorer) and enforce an app-size budget in CI; size wins also improve startup.

Revision Notes

- App Bundle (`flutter build appbundle`) → Play per-device APK (ABI/density/resources); iOS app thinning/slicing; (APK fat → `--split-per-abi`).
- Flags: `--release` (AOT + tree-shaking + `--tree-shake-icons`), `--obfuscate --split-debug-info=<dir>` (smaller + secure; ARCHIVE mapping for crash de-symbolication).
- Assets (biggest content win): compress/right-size/WebP, subset fonts, CDN+cache large media, remove unused; deps hygiene; R8/resource shrinking (Android); defer heavy/rare features (deferred loading).
- Measure: `--analyze-size` + App Bundle Explorer/DevTools app-size; set + CI-enforce an app-size budget; size ↔ startup overlap.

Practice Questions

1. Why does an App Bundle reduce the download vs an APK?
2. What are the main asset-size levers?
3. How do you measure and budget app size?

Coding Questions

1. Build a release App Bundle with obfuscation + split-debug-info and archive the mapping.
2. Optimize an oversized image/font asset and remove an unused dependency.
3. Add a CI step measuring app size and failing on a budget regression.

Mini Project

App-size optimization (Flutter/deployment): Take an app and reduce its size: publish an App Bundle (+ obfuscation + split-debug-info with archived mapping), optimize assets (compress/right-size images to WebP, subset fonts, tree-shake icons, remove unused assets/deps), enable Android resource shrinking, and defer one heavy feature — then measure with `--analyze-size` and add a CI app-size budget check. Acceptance: App Bundle + `--release --`

`obfuscate --split-debug-info` (mapping archived); assets optimized (biggest wins) + unused removed; deferred heavy feature; measured with `--analyze-size` (before/after); CI size-budget check; startup/conversion benefit noted.

Deployment Integration (Capstone: Full Release + Post-Launch)

> Assemble the last mile into a **release runbook**: **pre-release** (bump version + build number, build an **optimized signed** App Bundle/IPA, run the CI gate), **store** (complete/compliant listing + accurate privacy/rating), **submit** (with reviewer demo access), **release** (staged/phased rollout with **monitoring**), and **post-launch** (watch crash-free rate + reviews, respond to feedback, run a **forward-fix/hotfix** process). It ties CI/CD (Module 50), store setup, compliance, and size optimization into one repeatable process — and continues *after* launch, because shipping is the start of the operate/iterate loop, not the end.

Introduction

This module capstone composes the fundamentals, store setup/submission, review/compliance, and size optimization into one end-to-end release runbook, then covers the post-launch practices (monitoring, reviews, hotfixes, iteration) that make a launch sustainable. It's the "actually ship it and keep it healthy" deliverable.

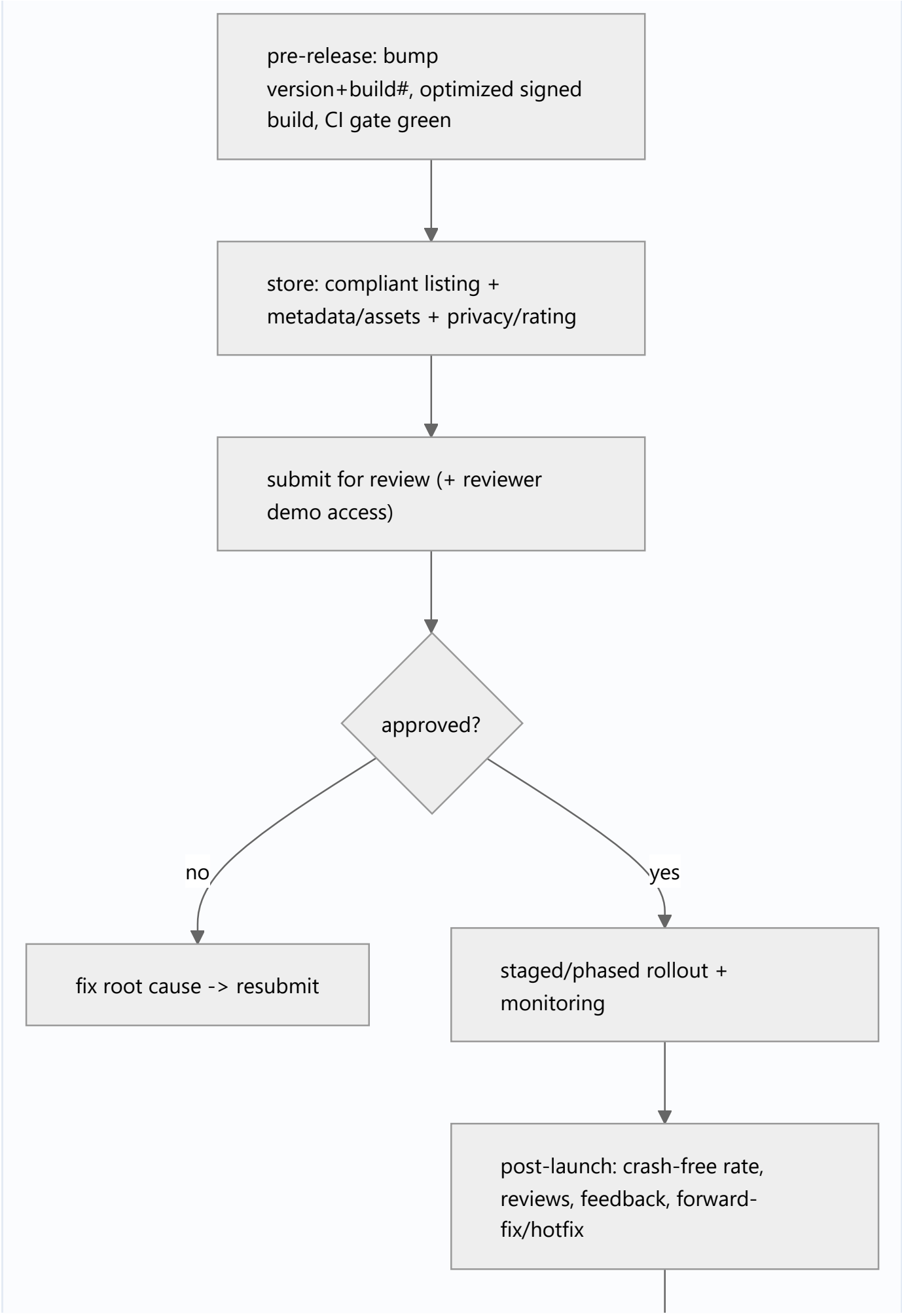
Why this concept exists

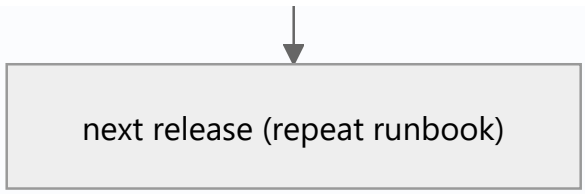
The pieces only deliver a **successful launch** when executed as one coordinated, repeatable process — and a launch isn't done at "approved": crashes surface at scale, users leave reviews, and regressions need fast fixes. A runbook + post-launch practices turn releasing from a stressful one-off into a reliable, iterative operation.

Real-world analogy

It's a **product launch playbook + store operations**: pre-launch QA + packaging (optimized signed build), a compliant shelf listing, passing inspection (review), a **phased regional rollout** you watch closely, and then **ongoing store operations** — reading customer reviews, tracking defect reports, and shipping quick fixes. The launch event is the beginning of running the product, not the finish line.

Internal Working


pre-release: bump version+build#, optimized signed build, CI gate green
store: compliant listing + metadata/assets + privacy/rating
submit for review (+ reviewer demo access)
approved?
no
yes
fix root cause -> resubmit
staged/phased rollout + monitoring
post-launch: crash-free rate, reviews, feedback, forward-fix/hotfix



- **Pre-release** (01_deployment_fundamentals.md/04_app_size_and_build_optimization.md/Module 50): bump **semver + unique build number**; build an **optimized, signed** App Bundle/IPA (`--release --obfuscate --split-debug-info`, App Bundle, optimized assets — **archive the symbol mapping**); ensure the **CI gate is green** (analyze/tests). Automate via CI/CD.
- **Store** (02_store_setup_and_submission.md): complete **listing** (metadata, correctly-sized/localized screenshots, feature graphic), **accurate compliance** (privacy policy, Data Safety/App Privacy matching behavior, content/age rating, permission declarations), pricing/availability.
- **Submit** (03_review_process_and_compliance.md): submit a **complete, crash-free** build with **reviewer demo credentials + notes**; budget review time (Apple); on rejection, **fix root cause + respond/resubmit**.
- **Release**: **staged/phased rollout** (5%→100% Play / Phased Release iOS) with **release notes**, watching **monitoring** (Module 52); **halt** if metrics degrade. Production is **gated** (mobile = Continuous Delivery — Module 50).
- **Post-launch (the loop that matters)**:
  - **Monitor** crash-free rate, errors, ANRs, performance (startup/frames), and adoption (Module 52) — de-symbolicate obfuscated crashes with the archived mapping.
  - **Reviews/ratings**: read + **respond** to store reviews (support + ASO signal); triage feedback into the backlog.
  - **Forward-fix/hotfix process**: mobile can't easily roll back → **staged rollout halt + feature flags/kill switches + expedited hotfix release**. Have this ready *before* launch.
  - **Iterate**: feed monitoring + feedback into the next release; run the **same runbook** each time (repeatable cadence).
- **A living runbook (deliverable)**: a checklist covering pre-release → store → submit → release → post-launch, so every release is consistent, automatable, and low-stress.
- **Right-sizing**: a solo app needs a lean runbook (build → store → submit → release → basic monitoring); a large app adds staged rollout, flags, dashboards, on-call, and formal hotfix SLAs. Scale to the product.

Memory Representation

Not runtime — a **runbook + release state**: version/build-number, artifact, listing/compliance status, review status, rollout %, and post-launch metrics/feedback. The invariant: every release follows the same checklist; launch transitions into an operate/iterate loop.

Compiler / Build Behavior

Pre-release produces an optimized, signed, versioned artifact (App Bundle/IPA) with an archived debug-info mapping; CI gates it. Deterministic via pinned toolchains.

Runtime Behavior

Users receive the release progressively (staged/phased); monitoring reports crashes/errors/performance at scale; hotfixes ship as expedited releases; feature flags toggle behavior server-side without a store round-trip.

Flutter Engine Behavior

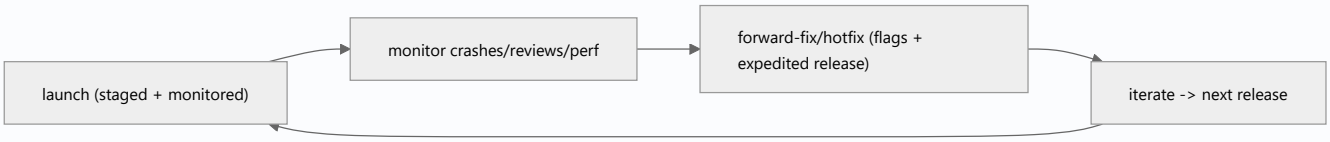
The optimized artifact runs per device (App Bundle/thinning); crash reporting captures engine/Dart errors for post-launch triage.

Dart VM Behavior

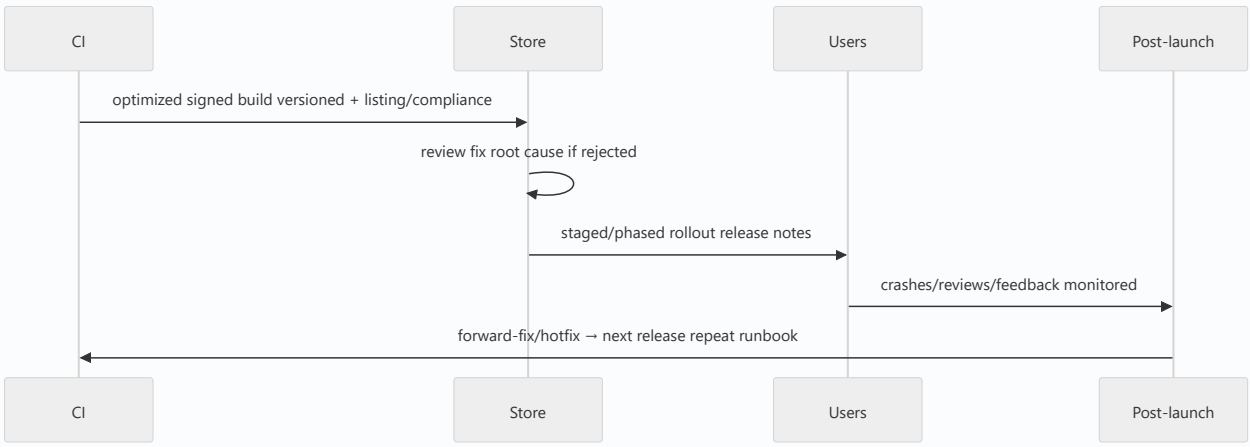
AOT release build in production; crash de-symbolication uses the archived mapping (04_app_size_and_build_optimization.md/Module 52).

Examples

```
Release runbook (checklist):
  PRE-RELEASE
    [ ] bump semver + unique build number (automated)
    [ ] optimized signed build (App Bundle/IPA, --release --obfuscate --split-debug-info) + archive mapping
    [ ] CI gate green (analyze + tests); E2E on critical journey
  STORE
    [ ] listing complete (metadata + screenshots/feature graphic, localized)
    [ ] compliance accurate (privacy policy, Data Safety/App Privacy, content rating, permissions)
  SUBMIT
    [ ] reviewer demo credentials + notes; budget review time
    [ ] on rejection: fix ROOT cause -> resubmit / respond in Resolution Center
  RELEASE
    [ ] release notes; staged/phased rollout %; monitoring watch; halt-if-degraded
  POST-LAUNCH
    [ ] monitor crash-free rate/errors/perf (de-symbolicate via mapping)
    [ ] read + respond to reviews; triage feedback -> backlog
    [ ] forward-fix/hotfix process ready (flags/kill switch + expedited release)
    [ ] iterate -> next release (repeat runbook)
```



Diagrams



Common Mistakes

Mistake	Why it's a problem	Fix
Treating launch as "done"	Crashes/feedback need response	Post-launch loop (monitor/respond/hotfix/iterate)
No monitoring after release	Blind to prod crashes	Crash-free/error/perf monitoring (Module 52)
Straight-to-100% release	Bad build hits everyone	Staged/phased rollout + halt
No hotfix/forward-fix process	Slow response to regressions	Flags/kill switch + expedited release ready pre-launch
Ad hoc, inconsistent releases	Errors, stress	A repeatable release runbook
Ignoring store reviews	Lost support + ASO signal	Read + respond; triage into backlog
Not archiving the symbol mapping	Unreadable prod crashes	Archive `--split-debug-info` per release
Over-engineering a solo app's process	Overhead	Right-size the runbook

Best Practices

- Run a **repeatable release runbook**: pre-release (versioned, **optimized signed** build, CI-gated, **mapping archived**) → compliant **store listing** → **submit** (reviewer demo access) → **staged/phased rollout + monitoring** (gated prod) → **post-launch loop**.
- **Monitor** crash-free rate/errors/perf (Module 52) and **de-symbolicate** with the archived mapping; **read + respond** to reviews; triage feedback.
- Have a **forward-fix/hotfix process ready before launch** (feature flags/kill switches + expedited release) since mobile rollback is hard; **halt** staged rollout on degradation.
- **Iterate** (feedback + metrics → next release) and **right-size** the runbook to the product; **automate** the pipeline steps via CI/CD (Module 50).

Performance

Post-launch monitoring surfaces real-world **performance** (startup/frames/crashes) at scale — closing the loop with performance budgets (Module 21/Module 47). Optimized builds + staged rollout protect install conversion + limit blast radius. A repeatable runbook improves **release velocity**.

Advantages / Disadvantages

- **+** Consistent low-stress releases, safe staged rollout, fast crash/feedback response, iterative improvement, automatable + right-sized.

- **−** Requires monitoring + hotfix infrastructure + process discipline; mobile rollback is hard (forward-fix); review latency; ongoing post-launch effort.

Interview Questions

1. 🟢 **What are the phases of a release runbook?** — Pre-release (version + optimized signed build + CI gate + archive mapping), store (compliant listing), submit (reviewer demo access), release (staged rollout + monitoring), post-launch (monitor/respond/hotfix/iterate).
2. 🟢 **Why isn't launch the end?** — Crashes surface at scale, users leave reviews, and regressions need fast fixes — launch begins the operate/iterate loop.
3. 🟡 **How do you handle a bad production release?** — Halt the staged rollout, toggle feature flags/kill switches, and ship an expedited forward-fix/hotfix (mobile can't easily roll back).
4. 🟡 **What post-launch signals do you monitor, and how?** — Crash-free rate, errors/ANRs, performance (startup/frames), adoption — via crash reporting/monitoring, de-symbolicating obfuscated crashes with the archived mapping.
5. 🟡 **Why respond to store reviews?** — Support/retention + ASO signal; triage feedback into the backlog for the next iteration.
6. 🔴 **Why archive the debug-info mapping at release time?** — To de-symbolicate obfuscated production crash reports; without it, prod stack traces are unreadable.
7. 🔴 **How do you right-size the release process?** — Solo/small: lean runbook + basic monitoring; large: staged rollout, flags, dashboards, on-call, hotfix SLAs — scale to the product.

Senior Engineer Tips

- Write the runbook once and follow it every release; consistency is what makes launches boring (good) instead of stressful firefights.
- Set up monitoring + a hotfix/feature-flag process before your first launch, and archive the symbol mapping per build; those are exactly what you'll wish you had the moment a production crash spikes.
- Always release staged + monitored and treat "approved" as the start of operations — reading reviews, watching crash-free rate, and shipping fast forward-fixes is where a launch actually succeeds or fails.

Architect Perspective

Deployment integration is the operate-and-iterate culmination: a repeatable runbook that turns CI/CD's signed optimized artifact into a compliant, reviewed, staged, monitored release — and then keeps the app healthy via monitoring, review response, and a forward-fix process. It closes the loop from build → ship → observe → fix → iterate, tying testing, CI/CD, size optimization, and

monitoring into one sustainable delivery practice. Launch is a phase in an ongoing operation, and the runbook + post-launch loop are what make that operation reliable (Module 50, Module 52, Module 47).

Summary

- Follow a repeatable release runbook: pre-release (version + optimized signed build + CI gate + archive mapping) → compliant store listing → submit (reviewer demo access) → staged/phased rollout + monitoring (gated prod) → post-launch.
- Post-launch = monitor crash-free/errors/perf (de-symbolicate via mapping), respond to reviews, run a forward-fix/hotfix process (flags + expedited release), iterate.
- Launch begins the operate/iterate loop; automate via CI/CD; right-size the runbook to the product.

Revision Notes

- Runbook: pre-release (bump semver + unique build#, optimized signed App Bundle/IPA `--release --obfuscate --split-debug-info` + archive mapping, CI gate) → store (compliant listing + accurate privacy/rating) → submit (reviewer demo creds/notes, budget review, fix root cause) → release (release notes, staged/phased rollout + monitoring, halt) → post-launch.
- Post-launch loop: monitor crash-free/errors/perf (de-symbolicate via mapping — Module 52), read + respond to reviews, forward-fix/hotfix ready (flags/kill switch + expedited release), iterate → next release.
- Gated prod (Continuous Delivery); mobile rollback hard → forward-fix; automate via CI/CD; right-size to product; launch = start of operate/iterate.

Practice Questions

1. What are the runbook phases from pre-release to post-launch?
2. How do you handle a bad release given hard mobile rollback?
3. What do you do after launch, and why isn't it the end?

Coding Questions

1. Write an end-to-end release checklist (pre-release → post-launch).
2. Define the post-launch monitoring + hotfix/forward-fix process.
3. Wire de-symbolication (archived mapping) into the crash-reporting flow.

Mini Project

Release runbook + post-launch (capstone — Flutter/deployment): Produce a full release runbook for an app: pre-release (auto version/build# bump, optimized signed App Bundle/IPA with obfuscation + archived mapping, CI gate), compliant store listing + submission (reviewer demo access), staged/phased rollout with monitoring, and a post-launch plan (crash-free/error/perf monitoring with de-symbolication, review response/triage, forward-fix/hotfix via flags + expedited release, iteration) — right-sized to the product and automated via CI/CD. Acceptance: repeatable runbook covering pre-release→store→submit→release→post-launch; optimized signed versioned build + archived mapping; compliance + reviewer access; staged rollout + monitoring + halt; post-launch monitoring/reviews/forward-fix/iterate; gated prod; right-sized + automatable.