50 · CI CD

Flutter Practice Notes

Contents

50 · CI/CD

CI/CD Fundamentals

CI Pipeline & Automation

Build Signing & Flavors/Environments

CD & Release Automation

CI/CD Integration (Capstone: Commit → Store Pipeline)

50 · CI/CD

Introduction

This module covers **Continuous Integration & Continuous Delivery/Deployment** for Flutter: the **fundamentals** (what CI/CD is, pipeline stages, principles), building a **CI pipeline** (automated analyze/test/build on every change — GitHub Actions/Codemagic, caching, monorepo/melos), **build signing & flavors/environments** (Android keystore, iOS certs/provisioning, Fastlane, dev/staging/prod flavors), and **CD release automation** (deploying to Play/App Store tracks, staged rollout, versioning), tied together in a capstone. It operationalizes testing (Module 49), deployment (Module 51), and the whole build/release workflow.

Why this module exists

Manual build/test/release is slow, error-prone, and doesn't scale: bugs slip in without automated checks, releases are risky rituals, and signing/store steps are fragile. **CI/CD automates the path from commit to production** — running analyze/test/build on every change (CI), and packaging/signing/releasing automatically (CD) — so teams ship **faster, safer, and repeatably**. It's the delivery backbone that makes testing enforceable, releases routine, and scaling teams possible.

Module map

#	File	Topic	Level
1	01_cicd_fundamentals.md	What CI/CD is, pipeline stages, principles	🔵
2	02_ci_pipeline_and_automation.md	CI: analyze/test/build automation, GH Actions/Codemagic, caching, monorepo	🔴
3	03_build_signing_and_flavors.md	Code signing (keystore/certs), flavors/environments, Fastlane	🔴
4	04_cd_release_automation.md	CD: store/track release, staged rollout, versioning	🔴
5	05_cicd_integration.md	Capstone: a full commit→store pipeline	🔴

> **Cross-references:** Testing (what CI runs): Module 49. Deployment/stores: Module 51. Monitoring (post-release): Module 52. Modular/monorepo (incremental CI): Module 45. Security (secrets/signing): Module 37. Native config (Android/iOS build): Module 27/Module 28.

Prerequisites

49 Testing, native build config (27/28), git/branching basics, 45 Modular Architecture (monorepo CI).

What you'll be able to do after this module

- Explain CI/CD, pipeline stages, and the principles behind them.
- Build a CI pipeline (analyze/test/build) with caching and (monorepo) incremental runs.
- Manage code signing (Android keystore, iOS certs) and dev/staging/prod flavors securely.
- Automate releases to Play/App Store tracks with staged rollout and versioning.
- Assemble a full commit→store pipeline with the right gates and secrets handling.

Capstone

Full pipeline: A CI/CD pipeline (GitHub Actions or Codemagic) that on PRs runs `flutter analyze` + tests (incremental via `melos --since` for a monorepo) with caching and merge gating; on tags/merges builds signed release artifacts per flavor (secure keystore/cert secrets), and deploys to internal/beta tracks (Play internal / TestFlight) with staged rollout + auto version bump — documented with the secrets/signing and branching strategy.

Summary

CI/CD automates commit→production: CI runs analyze/test/build on every change (fast feedback, merge gating); CD packages/signs/releases automatically (repeatable, staged). With caching, flavors, secure signing, and store automation (GH Actions/Codemagic/Fastlane), it lets teams ship faster and safer — the delivery backbone atop testing and deployment.

CI/CD Fundamentals

> **CI (Continuous Integration)** = every change is **automatically integrated + verified** (analyze/test/build) on a shared branch, frequently, so problems surface immediately. **CD** means either **Continuous Delivery** (every green build is **releasable**, deployed to a staging/beta track automatically, with a manual push to production) or **Continuous Deployment** (every green build goes **all the way to production** automatically). The pipeline is a sequence of **stages** (checkout → install → analyze → test → build → sign → deploy) where **each stage gates the next** — a failure stops the line. The principles: **automate everything, fast feedback, fail fast, and make it repeatable/reproducible**.

Introduction

This file defines CI, CD (delivery vs deployment), the pipeline-stage model, and the guiding principles — the conceptual frame before building actual pipelines. It clarifies the terminology teams constantly conflate.

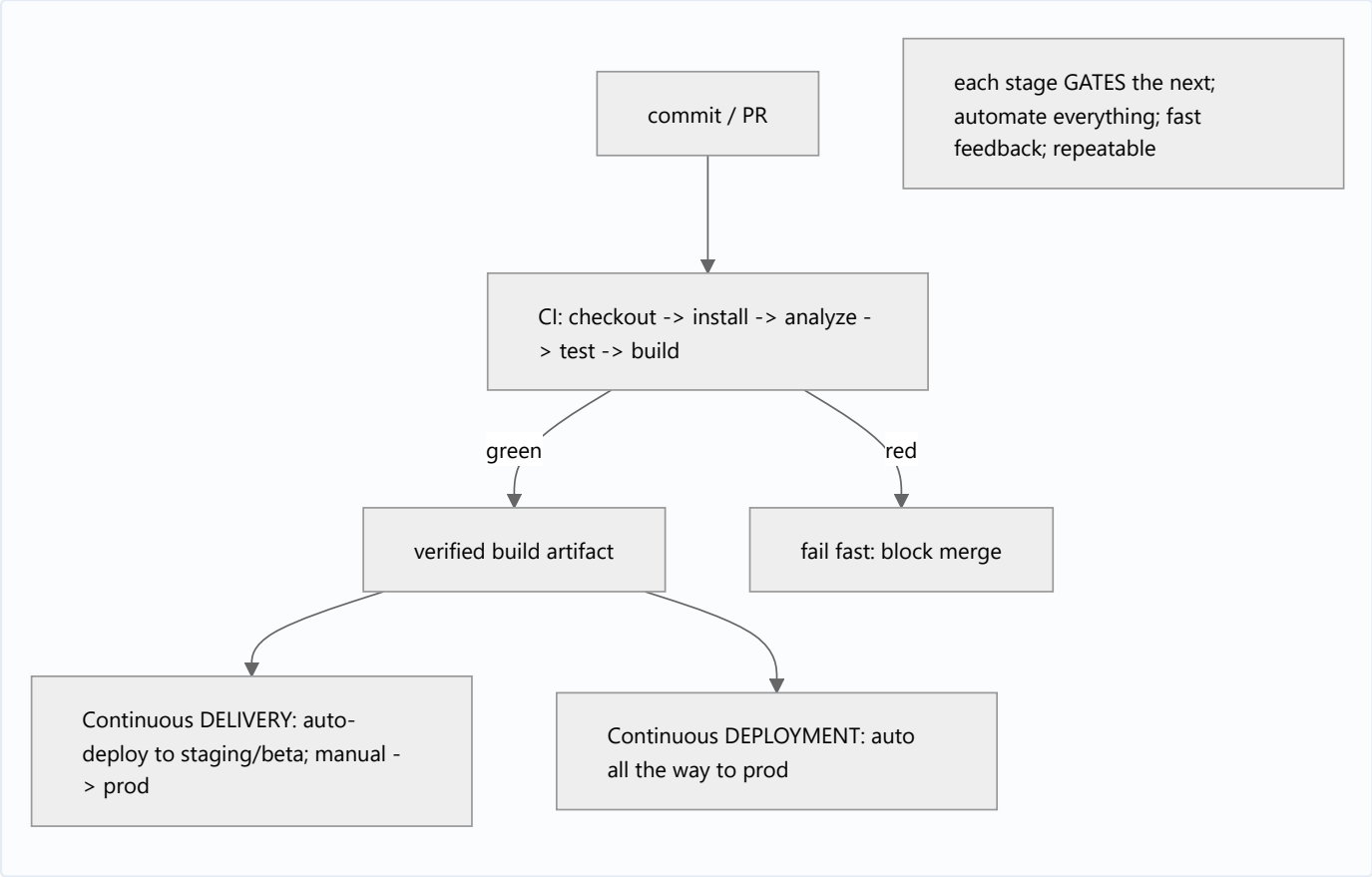
Why this concept exists

Manual integration + release is slow, inconsistent, and risky: bugs merge undetected, "works on my machine" diverges from production, and releases become fragile rituals. CI/CD encodes the build-test-release path as **automated, gated, repeatable** stages so integration problems surface in minutes and releases become routine — the foundation of modern delivery.

Real-world analogy

CI/CD is an **automated assembly line with quality gates**: each part is **checked as it's added** (CI — analyze/test), and a car only advances to the next station if it **passes inspection** (stage gating); a failed inspection **stops the line** (fail fast). **Continuous Delivery** = every finished car is ready to ship but a manager gives the final "ship it"; **Continuous Deployment** = finished cars roll straight to customers automatically. The line is **repeatable** — every car built the same way.

Internal Working



- **Continuous Integration (CI)**: developers **integrate frequently** (small, frequent merges) into a shared branch; **every change triggers an automated pipeline** (analyze/lint → test → build) that **verifies** it. Goal: catch integration problems **immediately**, keep the main branch **always green + buildable**.
- **Continuous Delivery vs Deployment (CD — know the difference)**:
  - **Continuous Delivery**: every green build is **automatically made releasable** and deployed to a **staging/beta** environment; **promotion to production is a manual (one-click) decision**. Most mobile teams do this (store review + business timing).
  - **Continuous Deployment**: every green build goes **automatically to production** — no manual gate. Common on the web/backend; **rarer for mobile** (app-store review, release cadence) but achievable for staged rollouts.
- **Pipeline stages (each gates the next)**: **checkout** → **install deps** (`pub get`) → **static analysis** (`flutter analyze` /lint) → **test** (unit/widget, then integration) → **build** (per platform/flavor) → **sign** → **deploy** (track/store). A failure at any stage **stops the pipeline** (fail fast) — no bad artifact proceeds.
- **Triggers**: pipelines run on events — **PR** (analyze + fast tests), **push to main/merge** (full build), **tag/release** (build + sign + deploy), **schedule** (nightly E2E). Different triggers → different stages (02_ci_pipeline_and_automation.md).
- **Core principles**:

    - **Automate everything** (build/test/sign/release) — no manual, error-prone steps.
    - **Fast feedback**: PR pipelines must be **quick** (analyze + unit/widget) so devs learn in minutes; heavy stages (E2E, full builds) run later/less often.
    - **Fail fast**: stop on the first failure; surface it clearly.
    - **Repeatable/reproducible**: same inputs → same result (pinned tool versions, clean environments, no hidden local state) — kills "works on my machine."
    - **Merge gating**: a PR can't merge unless the pipeline is green (protected branches).
    - **Small, frequent integrations**: reduce merge pain + risk.
- **Tools (overview)**: **GitHub Actions**/**GitLab CI** (general, config-as-code), **Codemagic**/**Bitrise** (Flutter/mobile-specialized: signing, store deploy built-in), **Fastlane** (build/sign/release automation, often invoked by the above). Choice is covered in 02_ci_pipeline_and_automation.md/04_cd_release_automation.md.
- **Why it matters at scale**: CI/CD makes **testing enforceable** (gate), **releases routine** (automated), and **team scaling possible** (parallel work verified continuously — Module 47).

Memory Representation

Not runtime — a **pipeline definition** (config-as-code: stages, triggers, gates) + a run history (each commit's pipeline result). The invariant: main is always green + buildable; each stage gates the next.

Compiler Behavior

Pipelines invoke the compiler/build (`flutter build` , `analyze`); reproducibility requires **pinned tool versions** (Flutter/Dart SDK) so builds are deterministic across environments.

Runtime Behavior

Stages run sequentially (or with parallel jobs), gating on exit codes; a non-zero exit stops the pipeline. Delivery deploys to staging automatically; deployment continues to production automatically.

Flutter Engine Behavior

Not applicable (pipelines build/test Flutter; they don't run its engine specially — except integration tests on emulators — Module 49).

Dart VM Behavior

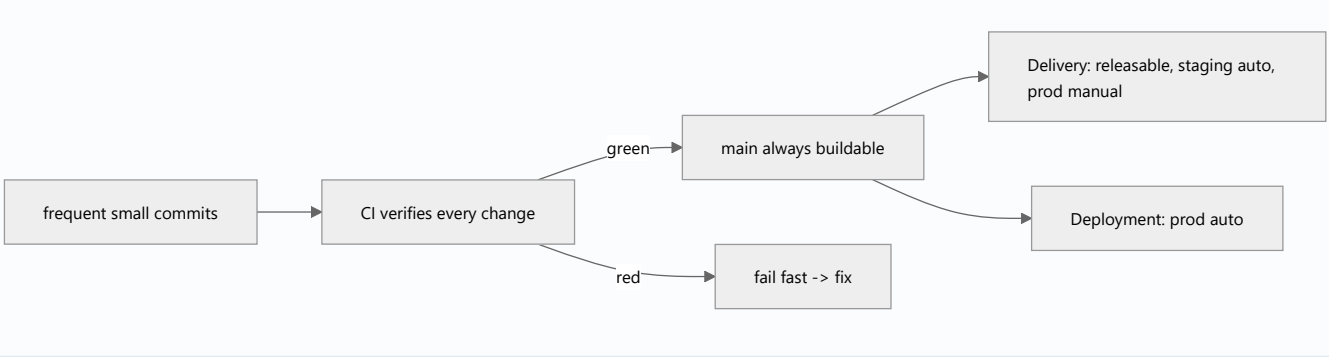
Not applicable beyond running Dart tests/builds; incremental/monorepo builds speed pipelines (Module 45).

Examples

```
Pipeline stages (each gates the next):
  checkout -> pub get -> flutter analyze -> flutter test -> flutter build (flavor) -> sign -> deploy
  (fail at any stage -> STOP the pipeline)

Trigger -> stages:
  PR              -> analyze + unit/widget tests            (fast feedback, merge gate)
  merge to main   -> + build signed artifact + deploy beta  (Continuous Delivery)
  tag v1.2.0      -> + deploy to production track (staged)   (Delivery: manual promote / Deployment: auto)
  nightly         -> integration/E2E on emulators
```

```
Delivery vs Deployment:
  Continuous Delivery:   green build -> auto to staging/beta -> MANUAL one-click to prod  (typical mobile)
  Continuous Deployment: green build -> AUTO all the way to prod                          (rarer for
mobile)
```

Diagrams



Common Mistakes

Mistake	Why it's wrong	Fix
Conflating Delivery and Deployment	Different automation of prod	Delivery = manual prod gate; Deployment = auto prod
No merge gating	Bad code lands on main	Protected branch requires green pipeline
Slow PR pipelines	Poor feedback, devs bypass	Fast stages on PR; heavy stages later
Non-reproducible builds	"Works on my machine"	Pin tool versions, clean environments
Manual build/sign/release steps	Error-prone, not scalable	Automate everything
One giant stage (no gating)	Can't fail fast/pinpoint	Distinct gated stages
Rare, big integrations	Merge hell, late bugs	Small, frequent integrations

Best Practices

- **Integrate frequently** (small merges) and **verify every change** with an automated pipeline; keep **main always green + buildable** via **merge gating**.
- Model the pipeline as **gated stages** (checkout → install → analyze → test → build → sign → deploy) that **fail fast**; run **fast stages on PRs** (analyze + unit/widget) and **heavy stages** (E2E, full builds/deploy) on merge/tag/nightly.
- **Automate everything** (build/test/sign/release) and make it **reproducible** (pinned tool versions, clean environments).
- Choose **Delivery vs Deployment** deliberately (mobile usually Delivery: auto-to-staging, manual prod); pick tools (GH Actions/Codemagic/Fastlane) to fit.

Performance

Pipeline speed is the KPI: fast PR feedback (analyze + unit/widget in minutes) keeps developers in flow; caching + incremental/monorepo runs keep CI fast at scale (02_ci_pipeline_and_automation.md/Module 45). Slow pipelines get bypassed; fast, reliable ones get trusted.

Advantages / Disadvantages

- **+** Immediate integration feedback, always-green main, repeatable + automated builds/releases, faster + safer shipping, enables team scaling.
- **−** Setup + maintenance (pipelines, secrets, signing), CI infra cost/time, requires discipline (frequent small merges, green-before-merge), mobile prod-gate nuances.

Interview Questions

1. 🟢 **What is Continuous Integration?** — Frequently integrating changes into a shared branch, each automatically verified (analyze/test/build), so integration problems surface immediately and main stays green.
2. 🟢 **Continuous Delivery vs Continuous Deployment?** — Delivery: every green build is releasable + auto-deployed to staging, with a manual push to prod; Deployment: every green build goes automatically to prod.
3. 🟡 **What are the typical pipeline stages, and how do they relate?** — checkout → install → analyze → test → build → sign → deploy; each gates the next, and a failure fails fast (stops the line).
4. 🟡 **Why do different triggers run different stages?** — Fast feedback: PRs run fast stages (analyze + unit/widget); merges/tags run heavy stages (full build, deploy); nightly runs E2E — keeping PR pipelines quick.
5. 🟡 **What makes a pipeline reproducible, and why care?** — Pinned tool versions + clean environments + no hidden local state → deterministic builds, eliminating "works on my machine."
6. 🔴 **Why is Continuous Deployment rarer for mobile?** — App-store review, release cadence, and business timing usually require a manual production gate (Delivery), though staged rollouts approximate automation.
7. 🔴 **What are the core CI/CD principles?** — Automate everything, fast feedback, fail fast, reproducibility, merge gating, and small frequent integrations.

Senior Engineer Tips

- Get the terminology right and design intent accordingly: most mobile teams do Continuous Delivery (auto-to-staging, manual prod), not full Continuous Deployment — say which you mean.
- Keep PR pipelines fast (analyze + unit/widget) and push heavy work (E2E, full builds, deploy) to merge/tag/nightly; a slow PR pipeline is the fastest way to get developers to route around CI.
- Make builds reproducible (pin the Flutter/Dart SDK, clean runners) from day one; non-deterministic CI is a trust-destroyer that surfaces at the worst time.

Architect Perspective

CI/CD is the delivery backbone: it turns the build-test-release path into automated, gated, reproducible stages so integration problems surface instantly and releases become routine. The fundamentals — automate everything, fast feedback, fail fast, reproducibility, merge gating — are what make testing enforceable, releases safe, and team scaling viable. Getting the stage model +

Delivery-vs-Deployment intent right sets up the concrete pipeline, signing, and release automation that follow (02_ci_pipeline_and_automation.md, 04_cd_release_automation.md, Module 49, Module 47).

Summary

- CI = auto-integrate + verify every change (analyze/test/build), keep main green; CD = Delivery (auto-to-staging, manual prod) or Deployment (auto-to-prod).
- Pipeline = gated stages (checkout→install→analyze→test→build→sign→deploy) that fail fast; different triggers (PR/merge/tag/nightly) run different stages.
- Principles: automate everything, fast feedback, fail fast, reproducible, merge-gated, small frequent integrations.

Revision Notes

- CI = frequent integration + automated verify (analyze/test/build), main always green + gated merges. CD: Delivery (green→staging auto, prod manual — typical mobile) vs Deployment (green→prod auto).
- Stages (gate next, fail fast): checkout → pub get → analyze → test → build (flavor) → sign → deploy. Triggers→stages: PR (fast: analyze+unit/widget), merge (build+beta), tag (prod/staged), nightly (E2E).
- Principles: automate everything, fast feedback, fail fast, reproducible (pin SDK/clean env), merge gating, small frequent integrations. Tools: GH Actions/GitLab CI (general), Codemagic/Bitrise (mobile), Fastlane (build/sign/release).

Practice Questions

1. Distinguish Continuous Delivery from Continuous Deployment.
2. What are the pipeline stages, and what does "each gates the next" mean?
3. Why run different stages on PR vs tag vs nightly?

Coding Questions

1. List the stages for a Flutter PR pipeline vs a release pipeline.
2. Define which triggers run which stages for a mobile app.
3. Identify a non-reproducible-build risk and how to fix it.

Mini Project

Pipeline design (Flutter/CI-CD): Design (on paper) a CI/CD pipeline: define the stages (checkout→install→analyze→test→build→sign→deploy), map triggers to stages (PR fast feedback, merge→beta, tag→prod, nightly→E2E), decide Delivery vs Deployment (with the prod gate), and list the principles you're applying (fast feedback, fail fast, reproducibility, merge gating).
Acceptance: gated stages with fail-fast; trigger→stage mapping (fast PR, heavier later); Delivery-vs-Deployment decision justified; reproducibility + merge-gating addressed; principles applied.

CI Pipeline & Automation

> A Flutter CI pipeline is **config-as-code** (a YAML workflow in GitHub Actions/GitLab CI, or a Codemagic/Bitrise config) that, on each trigger, **sets up the environment** (checkout, pinned Flutter SDK), **installs deps** (`pub get`), and runs the **verification stages** — `flutter analyze` → `flutter test` **(+ coverage)** → `flutter build` (per flavor) → optionally **integration tests on emulators**. The levers that make it fast + reliable: **dependency + build caching**, **parallel jobs** (Android/iOS/web in matrix), **incremental runs** in a monorepo (`melos run test --since`), and **merge gating** on green. Keep **PR pipelines fast** (analyze + unit/widget); push heavy work (E2E, full builds) to merge/nightly.

Introduction

This file covers building the CI half concretely: the YAML/config structure, the analyze/test/build stages, caching + parallelism + matrices, monorepo incremental CI, and merge gating — with GitHub Actions as the primary example and Codemagic as the mobile-specialized alternative. It implements the fundamentals (01_cicd_fundamentals.md) and runs the tests from Module 49.

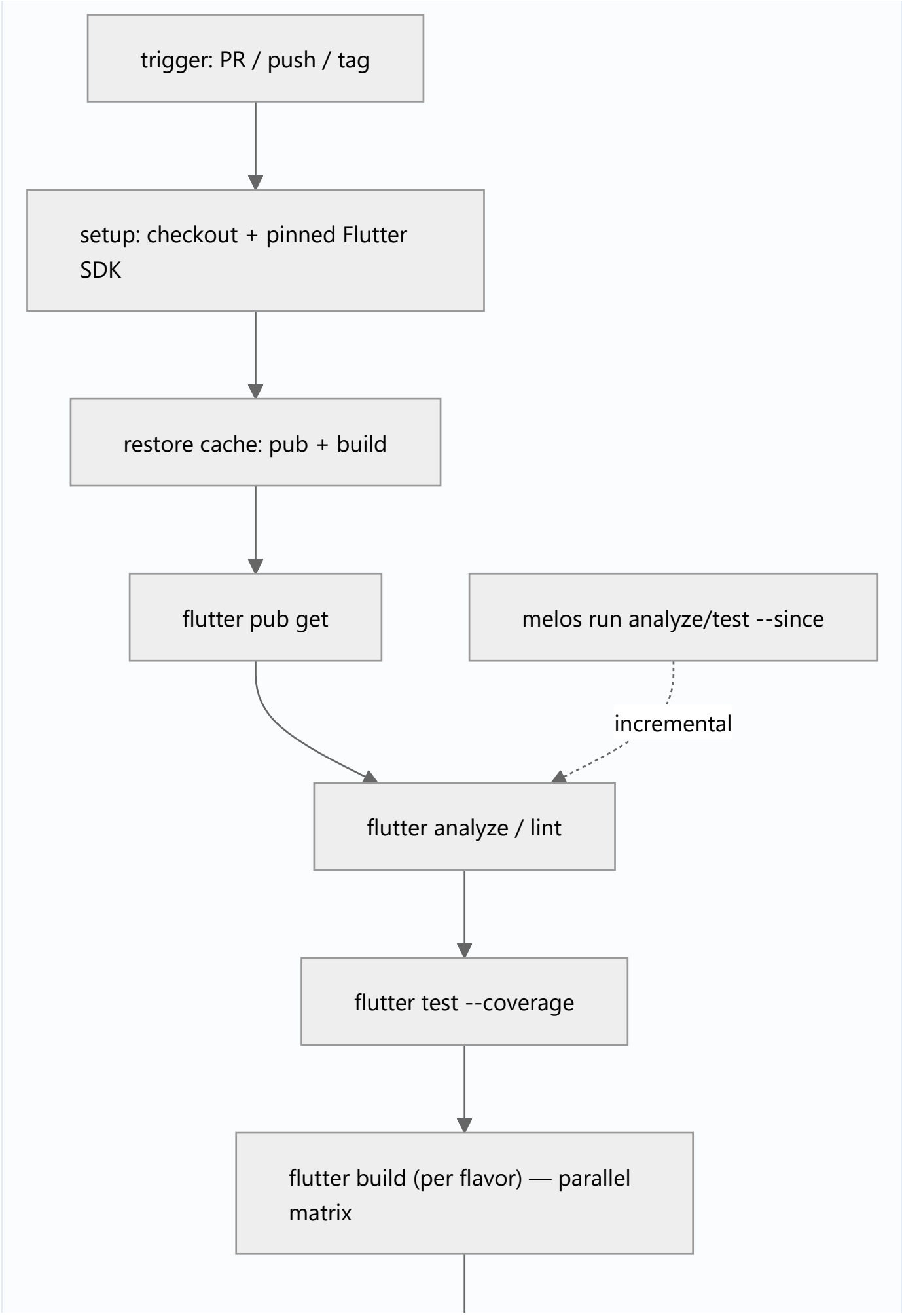
Why this concept exists

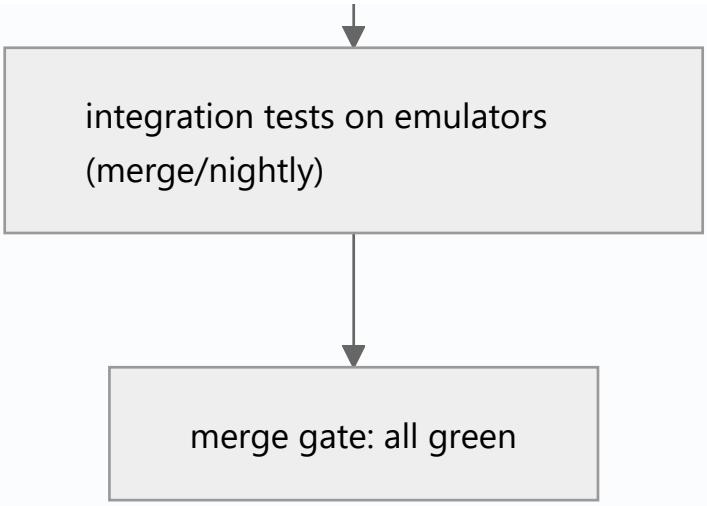
CI's value depends on being **fast, reliable, and comprehensive** — a slow or flaky pipeline gets bypassed; a shallow one misses regressions. Knowing how to structure the workflow, cache aggressively, parallelize, and run incrementally is what turns "we have CI" into "CI gives trusted feedback in minutes." It's also where testing becomes **enforced** (gate) rather than optional.

Real-world analogy

The CI pipeline is a **standardized inspection station** on the assembly line: it always uses the **same certified tools** (pinned SDK), keeps **frequently-used parts on hand** (caching — no re-fetching), runs **multiple inspectors in parallel** (matrix jobs for Android/iOS), inspects **only the changed sections** when possible (incremental/monorepo), and **won't let a car pass** without clearing inspection (merge gating). A slow, single-inspector station bottlenecks the whole factory.

Internal Working


trigger: PR / push / tag
setup: checkout + pinned Flutter SDK
restore cache: pub + build
flutter pub get
melos run analyze/test --since
incremental
flutter analyze / lint
flutter test --coverage
flutter build (per flavor) — parallel matrix



- **Config-as-code**: the pipeline lives in the repo — **GitHub Actions** (`.github/workflows/*.yml`), **GitLab CI** (`.gitlab-ci.yml`), or a mobile-specialized service (**Codemagic**/**Bitrise**) config. Versioned, reviewable, reproducible.
- **Environment setup**: **checkout** the repo, **install a pinned Flutter/Dart SDK** (e.g., `subosito/flutter-action` with a fixed version) — pinning ensures reproducibility.
- **Verification stages** (from Module 49):
  - `flutter pub get` (install deps).
  - `flutter analyze` (+ `dart format --set-exit-if-changed`) — static analysis/lint gate.
  - `flutter test --coverage` — unit + widget tests (upload LCOV to Codecov etc.).
  - `flutter build apk/appbundle/ipa/web` per **flavor** — verifies it builds (release build on tags).
  - `flutter test integration_test` on **emulators/device farm** (Firebase Test Lab) — heavier, run on **merge/nightly**, not every PR.
- **Caching (biggest speed lever)**: cache the **pub cache** (`~/.pub-cache`), **Gradle** (`~/.gradle`), **CocoaPods** (`Pods/`), and build outputs — keyed by lockfiles (`pubspec.lock` , etc.). Cache hits skip re-downloading deps → pipelines go from many minutes to a fraction.
- **Parallelism + matrices**: run **Android/iOS/web builds and test shards in parallel jobs** (a matrix), so total wall-clock ≈ the slowest job, not the sum. Split large test suites into shards.
- **Monorepo incremental CI** (Module 45): with melos, `melos bootstrap` then `melos run analyze/test --since=origin/main` runs only **changed packages (+ dependents)** — CI scales with change size, not repo size (requires stable core/contracts).
- **Fast-PR discipline**: PR pipeline = **analyze + unit/widget (cached, parallel)** for minutes-scale feedback; **defer** E2E + full multi-platform release builds to merge/tag/nightly. Slow PR pipelines get bypassed.
- **Merge gating**: **protected branches require the pipeline green** to merge (+ required reviewers/CODEOWNERS — Module 47); optionally a **coverage threshold/no-regression** check.

- **Secrets** (for signed builds/integration with services): stored in the CI provider's **encrypted secrets** (never in the repo), injected as env vars (03_build_signing_and_flavors.md/Module 37).
- **GH Actions vs Codemagic**: **GH Actions** = general, flexible, free tier, but you assemble signing/store steps yourself; **Codemagic/Bitrise** = Flutter/mobile-first with **built-in signing + store publishing** (less setup, paid). Choose by team needs.

Memory Representation

Not runtime — a **workflow definition** (jobs/steps/triggers/caches/matrix) + per-run logs/artifacts. Caches persist between runs keyed by lockfiles; artifacts (builds, coverage) are stored per run.

Compiler Behavior

The pipeline invokes the Flutter compiler/analyzer; pinned SDK versions make compilation reproducible; caches speed dependency resolution + incremental compilation.

Runtime Behavior

Jobs run on ephemeral CI runners (clean each time — reproducible); stages gate on exit codes; parallel jobs run concurrently; caches restore/save around runs. Integration tests spin up emulators.

Flutter Engine Behavior

Integration-test jobs run the engine on emulators (Android)/simulators; other stages don't run the engine (Module 49).

Dart VM Behavior

Analyze/test run in the VM; incremental monorepo runs recompile only changed packages (Module 45).

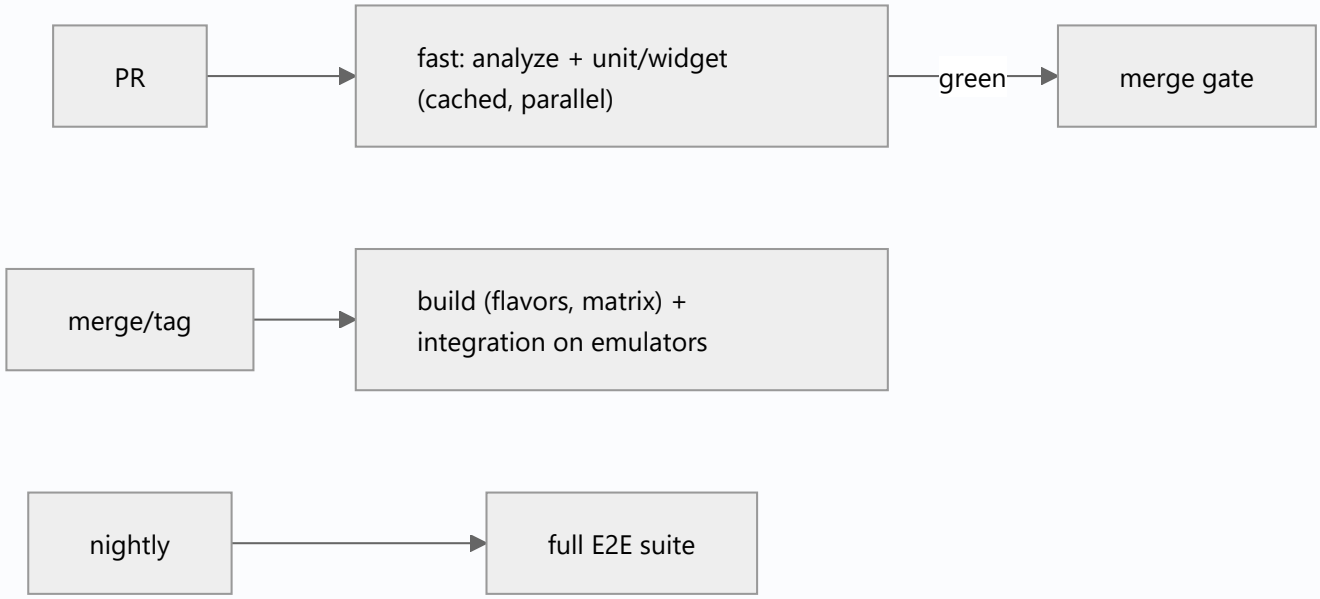
Examples

```yaml
# .github/workflows/ci.yml — fast PR pipeline (analyze + test), cached
name: CI
on: [pull_request]
jobs:
  verify:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: subosito/flutter-action@v2
        with: { flutter-version: '3.24.0', cache: true }   # pinned + cache SDK
      - run: flutter pub get
      - run: dart format --set-exit-if-changed .            # format gate
      - run: flutter analyze                                # analyze gate
      - run: flutter test --coverage                        # unit + widget (+LCOV)
      # (build/integration deferred to merge/nightly workflows)
```

```yaml
# Parallel build matrix (Android/iOS/web) + monorepo incremental tests
strategy:
  matrix: { platform: [apk, ipa, web] }
steps:
  - run: flutter build ${{ matrix.platform }} --release --flavor prod
# Monorepo (melos): only changed packages
  - run: melos bootstrap
  - run: melos run test --since=origin/main
```

```
Caching keys (skip re-downloading deps):
  ~/.pub-cache   keyed by pubspec.lock
  ~/.gradle      keyed by *.gradle / gradle-wrapper
  ios/Pods       keyed by Podfile.lock
  -> cache HIT turns a multi-minute install into seconds
```

Diagrams



Common Mistakes

Mistake	Why it's a problem	Fix
No caching	Slow pipelines (re-download deps)	Cache pub/gradle/pods keyed by lockfiles
Running E2E/full builds on every PR	Slow feedback	PR = fast (analyze+unit/widget); heavy on merge/nightly
Unpinned Flutter SDK	Non-reproducible builds	Pin SDK version
Serial single job	Long wall-clock	Parallel jobs/matrix + shards
Full monorepo test every run	Doesn't scale	`melos run --since` (changed packages)
No merge gating	Regressions land	Protected branch requires green
Secrets in the repo/config	Security breach	CI encrypted secrets (env vars)
No format/analyze gate	Style/quality drift	`analyze` + `format --set-exit-if-changed`

Best Practices

- Define CI as **config-as-code** with a **pinned Flutter SDK**; run **analyze + format +** `flutter test --coverage` on PRs, deferring **E2E + full/multi-platform builds** to merge/nightly (fast feedback).
- **Cache aggressively** (pub/gradle/pods keyed by lockfiles) and **parallelize** (Android/iOS/web matrix + test shards); in monorepos use `melos run --since` for incremental runs.

- **Gate merges** on green (protected branches + CODEOWNERS, optional coverage threshold); store **secrets in the CI provider's encrypted store** (never in the repo).
- Choose the platform to fit (**GH Actions/GitLab** general vs **Codemagic/Bitrise** mobile-specialized with built-in signing/publishing); keep runners **clean/ephemeral** for reproducibility.

Performance

Caching + parallelism + incremental (`melos --since`) are the speed levers that keep CI in the minutes range as the codebase grows — the difference between trusted, fast feedback and a bypassed bottleneck. Fast-PR discipline (heavy stages later) keeps the inner loop tight (Module 47).

Advantages / Disadvantages

- **+** Fast trusted feedback, enforced testing (gate), reproducible builds, parallel/incremental scaling, config-as-code (reviewable/versioned).
- **−** Setup + maintenance (YAML/caches/matrices), CI-minutes cost, secrets/signing config, emulator flakiness for E2E, provider lock-in nuances.

Interview Questions

1. 🟢 **What stages does a Flutter CI pipeline run?** — checkout + pinned SDK → `pub get` → `analyze`/format → `flutter test --coverage` → `flutter build` (per flavor) → integration tests (on merge/nightly).
2. 🟢 **How do you keep CI fast?** — Cache deps (pub/gradle/pods), parallelize (matrix + shards), run incrementally in monorepos (`melos --since`), and keep heavy stages (E2E/full builds) off the PR path.
3. 🟡 **Why pin the Flutter SDK version?** — Reproducibility — the same commit builds identically across runs/machines, avoiding "works on my machine."
4. 🟡 **What runs on a PR vs on merge/nightly?** — PR: analyze + unit/widget (fast); merge/tag: builds + integration; nightly: full E2E — for fast feedback.
5. 🟡 **How does monorepo CI scale?** — `melos run analyze/test --since=origin/main` runs only changed packages (+ dependents), so CI cost scales with change size (given stable roots).
6. 🔴 **How are secrets handled in CI?** — Stored in the provider's encrypted secrets, injected as env vars at runtime — never committed to the repo/config.
7. 🔴 **GitHub Actions vs Codemagic — trade-offs?** — GH Actions: general, flexible, cheap, but you build signing/store steps yourself; Codemagic/Bitrise: Flutter/mobile-first with built-in signing/publishing (less setup, paid).

Senior Engineer Tips

- Cache the pub/gradle/pods directories keyed by lockfiles and parallelize platform builds first; those two changes usually cut pipeline time the most and are what make CI trusted rather than bypassed.
- Keep the PR pipeline ruthlessly fast (analyze + unit/widget) and shove E2E + full/multi-platform builds to merge/nightly; developers judge CI by PR latency.
- In monorepos, wire `melos --since` early so CI stays fast as the repo grows — but keep core/contracts stable, or every change rebuilds everything.

Architect Perspective

The CI pipeline is where testing becomes enforced and feedback becomes fast: config-as-code stages with pinned SDKs, aggressive caching, parallel matrices, and incremental monorepo runs, gated at merge. Designed well, it gives minutes-scale trusted verification that scales with the codebase — the foundation the CD/signing/release layers build on and a core lever of scalable, multi-team delivery (01_cicd_fundamentals.md, 03_build_signing_and_flavors.md, Module 45, Module 47).

Summary

- CI = config-as-code (GH Actions/Codemagic): pinned SDK → `pub get` → analyze/format → `flutter test --coverage` → build (flavors) → integration (merge/nightly), gated at merge.
- Fast + reliable via caching (pub/gradle/pods), parallel matrices/shards, and monorepo incremental runs (`melos --since`); keep PR pipelines fast, defer heavy stages.
- Secrets in encrypted CI store; choose GH Actions (general) vs Codemagic/Bitrise (mobile-specialized) by needs.

Revision Notes

- Config-as-code (GH Actions `.github/workflows` , GitLab CI, Codemagic/Bitrise); setup = checkout + pinned Flutter SDK (`subosito/flutter-action`).
- Stages: `pub get` → `dart format --set-exit-if-changed` + `flutter analyze` → `flutter test --coverage` → `flutter build` (flavor, matrix) → `integration_test` on emulators (merge/nightly).
- Speed: cache pub/gradle/pods (keyed by lockfiles), parallel matrix + shards, monorepo `melos run --since` . PR fast (analyze+unit/widget), heavy later. Merge gate (protected + CODEOWNERS + coverage). Secrets in encrypted CI store. GH Actions (general) vs Codemagic/Bitrise (mobile signing/publish built-in).

Practice Questions

1. What runs on a PR pipeline vs merge/nightly, and why?
2. What are the biggest levers for CI speed?
3. How do secrets get into a CI build safely?

Coding Questions

1. Write a GitHub Actions workflow for a fast PR pipeline (analyze + test, cached).
2. Add a parallel build matrix (Android/iOS/web) + monorepo incremental tests.
3. Configure caching keyed by lockfiles for pub/gradle/pods.

Mini Project

CI pipeline (Flutter): Author a GitHub Actions (or Codemagic) CI config: a fast PR job (pinned SDK, cached pub/gradle/pods, `format` + `analyze` + `flutter test --coverage`), a merge/nightly job (parallel build matrix per platform/flavor + `integration_test` on an emulator), and (if monorepo) `melos run --since` incremental tests — with merge gating and secrets from the encrypted store. Acceptance: config-as-code with pinned SDK; PR pipeline fast (analyze+unit/widget, cached/parallel); heavy stages (E2E/full builds) on merge/nightly; caching keyed by lockfiles; monorepo incremental (if applicable); merge-gated; secrets encrypted (not in repo).

Build Signing & Flavors/Environments

> Release builds must be **code-signed** — **Android** with a **keystore** (private key; the upload/app-signing key that proves authorship) and **iOS** with **certificates + provisioning profiles** (via Apple Developer) — and CI must access these **secrets securely** (base64-encoded in the CI encrypted store, never in the repo). Apps also ship in multiple **flavors/environments** (dev/staging/prod) with **distinct bundle ids, config, API endpoints, and icons**, selected at build time (`--flavor` + `--dart-define` /entry point). **Fastlane** commonly automates the signing + build + upload steps that CI invokes.

Introduction

This file covers the two build-configuration pillars CI/CD needs: **signing** (Android keystore, iOS certs/provisioning, secure secret handling) and **flavors/environments** (dev/staging/prod separation), plus **Fastlane** as the automation glue. It builds on native config (Module 27/Module 28) and security (Module 37).

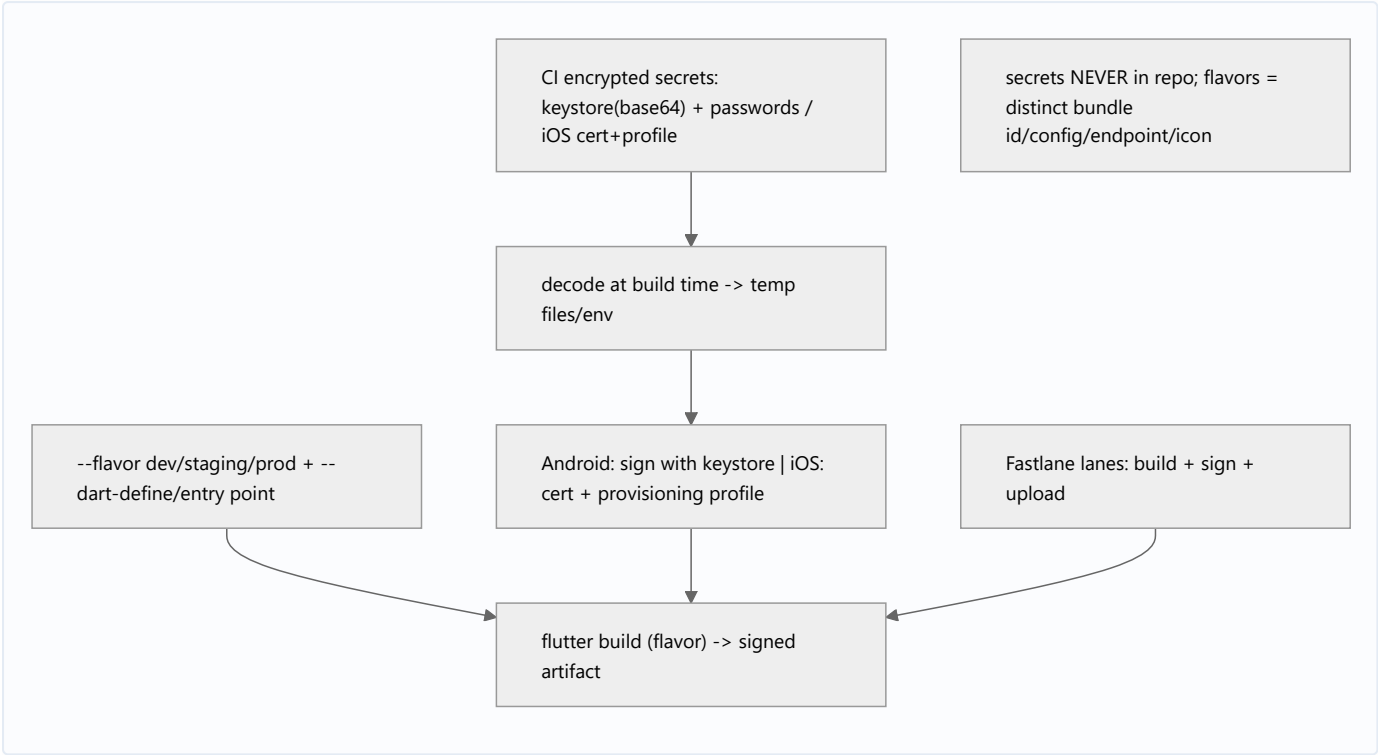
Why this concept exists

Stores only accept **signed** builds, and signing keys are **sensitive** (a leaked keystore lets others publish as you) — so CI must handle them securely. And teams need **separate environments** (dev/staging/prod) that coexist on a device with their own backends/config — requiring flavors. Automating these correctly (secrets + flavors + Fastlane) is what lets CI produce shippable, environment-correct, signed artifacts.

Real-world analogy

Signing is your **notarized seal + ID**: the keystore/cert proves *you* made this app; lose control of the seal and impostors can publish as you (guard it in a vault = encrypted secrets). **Flavors** are **product editions off one production line** — the "test kitchen" edition (dev), the "pre-launch" edition (staging), and the "retail" edition (prod) — same base, different labels/ingredients (bundle id/config/endpoint/icon), stamped at packaging time. **Fastlane** is the **automated packaging-and-shipping crew**.

Internal Working



- **Android signing**:
  - A **keystore** (`.jks` / `.keystore`) holds the **private signing key**. Configure `signingConfigs` / `buildTypes` in `android/app/build.gradle` reading credentials from `key.properties` (path/passwords/alias) — which is **git-ignored**.
  - **Play App Signing** (recommended): you sign with an **upload key**; Google holds/manages the **app signing key**. Protects against upload-key loss.
  - **In CI**: base64-encode the keystore + store it (plus passwords) in **CI encrypted secrets**; at build time **decode to a temp file** and set env/ `key.properties` , then `flutter build appbundle --release` . **Never commit the keystore/passwords.**
- **iOS signing** (more involved):
  - Needs a **signing certificate** (identifies the developer/team) + a **provisioning profile** (ties app id + devices + entitlements) from **Apple Developer**. Xcode uses these to sign.
  - **In CI**: store the cert (`.p12`) + profile (base64) + password in **encrypted secrets**; install them into a temporary keychain at build time (Fastlane **match**/ `sigh` / `cert` automate this), then `flutter build ipa` . `match` stores certs/profiles encrypted in a git repo for team sync.
  - Requires a **Mac runner** (macOS) for iOS builds.
- **Secret handling (security-critical)** (Module 37): keystores/certs/passwords/API keys live **only** in the CI provider's **encrypted secrets** (GitHub Secrets/Codemagic env), injected as env vars, decoded to **temp files** deleted after the build. **Never in the repo, logs, or config.**
- **Flavors / environments** (dev/staging/prod):

    - Each flavor has a **distinct application/bundle id** (`com.app.dev` , `com.app.staging` , `com.app`) so they **coexist** on a device, plus its own **app name/icon**, **API endpoint/config**, and Firebase config.
    - **Android**: `productFlavors` in `build.gradle` (+ per-flavor `google-services.json`). **iOS**: **schemes + xcconfig/build configs** (more manual — Module 28).
    - **Flutter side**: select config via `--flavor <name>` + `--dart-define` / `--dart-define-from-file` (compile-time config) or a **separate entry point** (`main_dev.dart`) — inject the environment (base URL, flags) at build time, **not** hardcoded.
    - Build: `flutter build appbundle --release --flavor prod --dart-define-from-file=env/prod.json` .
- **Fastlane** (automation glue): Ruby-based; **lanes** encapsulate build/sign/upload steps (`gym` / `build_app` for iOS build, `match` / `sigh` for signing, `supply` for Play upload, `pilot` for TestFlight). CI invokes Fastlane lanes; Codemagic/Bitrise offer built-in equivalents. Fastlane centralizes the fiddly signing/store steps.
- **Config-not-secret vs secret**: environment **config** (base URLs, flags) can be in `--dart-define` /env files (non-sensitive); **secrets** (keys/keystores) stay in the encrypted store — don't confuse the two.

Memory Representation

Not runtime — **build-time config + secrets**: keystore/certs (in encrypted secrets → temp files at build), flavor definitions (gradle/xcconfig), and per-flavor config (`--dart-define` /entry points). The signed artifact embeds the signature + flavor config.

Compiler / Build Behavior

Flutter/Gradle/Xcode compile per flavor + sign the release artifact using the provided key/cert; `--dart-define` values are compiled in; wrong/missing signing config fails the release build. Pinned toolchains keep it reproducible.

Runtime Behavior

The signed, flavored artifact runs pointing at its environment's config (endpoint/flags baked in); different flavors coexist (distinct ids). Signature is verified by the store/OS at install.

Flutter Engine Behavior

Not applicable (build/signing are native tooling); `--dart-define` values are available to Dart at runtime.

Dart VM Behavior

`--dart-define` / `--dart-define-from-file` inject compile-time constants (tree-shakable) — the idiomatic way to pass environment config without secrets in code.

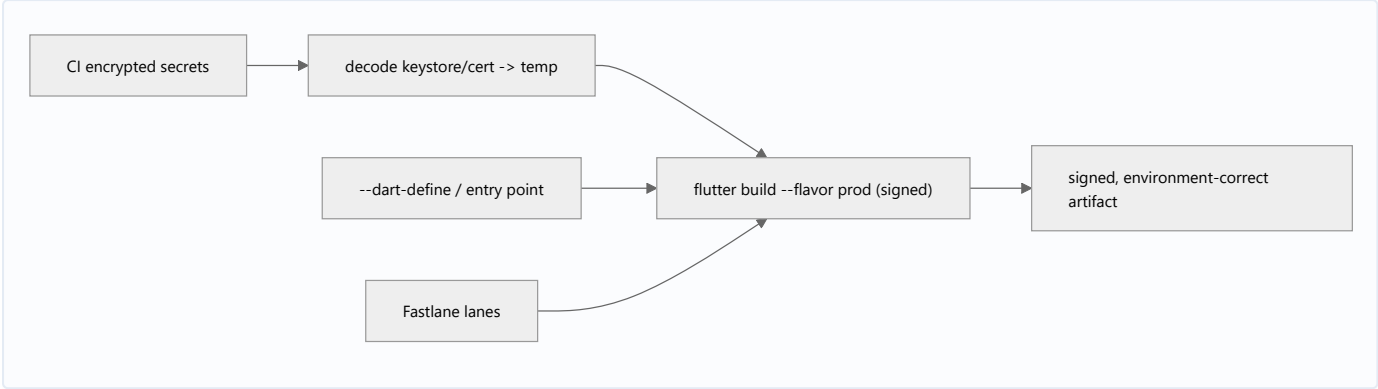
Examples

```
// android/app/build.gradle — signing from git-ignored key.properties + flavors
def keystoreProps = new Properties()
file('../key.properties').withInputStream { keystoreProps.load(it) }   // git-ignored
android {
  signingConfigs { release {
    storeFile file(keystoreProps['storeFile']); storePassword keystoreProps['storePassword']
    keyAlias keystoreProps['keyAlias']; keyPassword keystoreProps['keyPassword']
  } }
  buildTypes { release { signingConfig signingConfigs.release } }
  flavorDimensions "env"
  productFlavors {
    dev     { dimension "env"; applicationIdSuffix ".dev"; resValue "string","app_name","App Dev" }
    staging { dimension "env"; applicationIdSuffix ".staging" }
    prod    { dimension "env" }   // com.app
  }
}
```

```
# CI: decode keystore from secret at build time, then build a signed, flavored bundle
- run: echo "$KEYSTORE_BASE64" | base64 -d > android/app/upload.jks   # from CI secret
- run: |
    printf "storeFile=upload.jks\nstorePassword=$STORE_PW\nkeyAlias=$KEY_ALIAS\nkeyPassword=$KEY_PW\n" >
      android/key.properties
- run: flutter build appbundle --release --flavor prod --dart-define-from-file=env/prod.json
# (KEYSTORE_BASE64/STORE_PW/... come from GitHub encrypted secrets; files cleaned up after)
```

```
# Fastlane lane (iOS) — match handles certs/profiles, gym builds, pilot uploads to TestFlight
lane :beta do
  match(type: "appstore")          # sync signing cert + provisioning (encrypted git)
  build_app(scheme: "prod")        # gym: build signed .ipa
  upload_to_testflight             # pilot
end
```

Diagrams



Common Mistakes

Mistake	Why it's dangerous/wrong	Fix
Committing keystore/certs/passwords	Key theft → impostor publishing	Encrypted CI secrets only; git-ignore key files
Secrets in logs/ `--dart-define`	Leak	Secrets in encrypted store; only non-sensitive config in dart-define
No flavors (one bundle id)	Environments collide on device	Distinct bundle ids per flavor
Hardcoded API endpoints	Wrong env / rebuild churn	Inject via `--dart-define` /entry point
Not using Play App Signing	Upload-key loss = locked out	Enroll in Play App Signing (upload key)
Manual iOS signing in CI	Fragile/fails	Fastlane `match` / `sigh` + Mac runner
Missing per-flavor Firebase config	Wrong project	Per-flavor `google-services.json` / `GoogleService-Info.plist`
Confusing config with secrets	Over/under-protecting	Config (URLs/flags) in dart-define; keys in secrets

Best Practices

- **Store all signing material + passwords in the CI encrypted secrets** (base64 keystore, `.p12` /profiles), decode to **temp files at build time**, and **never** commit them or log them; git-ignore `key.properties`.
- Use **Play App Signing** (upload key) on Android and **Fastlane** `match` for iOS certs/profiles; build iOS on a **Mac runner**.
- Define **dev/staging/prod flavors** with **distinct bundle ids + config + endpoints + icons** (`productFlavors` /schemes) and inject environment via `--dart-define` **/entry point** (not hardcoded); per-flavor Firebase config.
- Automate build+sign+upload with **Fastlane lanes** (or Codemagic built-ins); keep **config (non-secret) separate from secrets**.

Performance

Not runtime perf; the concern is **build reliability + security**. Caching signing tooling/pods speeds iOS builds; per-flavor builds add matrix jobs. The real cost of getting this wrong is a **security incident** (leaked key) or a **broken release**, not slowness.

Advantages / Disadvantages

- **+** Shippable signed builds, secure secret handling, coexisting environments (dev/staging/prod), automated + reproducible signing/upload (Fastlane).
- **−** Signing setup is fiddly (esp. iOS certs/profiles), secret management discipline, flavor config per platform, Mac runner needed for iOS, Fastlane learning curve.

Interview Questions

1. 🟢 **How are Android and iOS release builds signed?** — Android with a keystore (private signing key; ideally Play App Signing with an upload key); iOS with a certificate + provisioning profile from Apple Developer.
2. 🟢 **How do you handle signing secrets in CI?** — Store them (base64 keystore/ `.p12` /profiles + passwords) in the CI encrypted secrets, decode to temp files at build time, and never commit or log them.
3. 🟡 **What are flavors/environments, and what differs per flavor?** — dev/staging/prod builds with distinct bundle ids (so they coexist), plus their own name/icon, API endpoint/config, and Firebase config.
4. 🟡 **How do you inject environment config into a Flutter build?** — `--flavor` + `--dart-define` / `--dart-define-from-file` or a separate entry point (`main_dev.dart`) — compile-time, not hardcoded; keep secrets out of dart-define.
5. 🟡 **What is Fastlane, and what does it automate?** — A Ruby tool with "lanes" automating build/sign/upload (`match` / `sigh` for signing, `gym` / `build_app` for build, `supply` / `pilot` for store upload) — invoked by CI.
6. 🔴 **Why use Play App Signing?** — You sign with an upload key while Google manages the app signing key — protecting against upload-key loss and easing key rotation.
7. 🔴 **What's the difference between environment config and secrets, and how are they handled?** — Config (base URLs/flags) is non-sensitive → `--dart-define` /env files; secrets (keys/keystores/passwords) → encrypted CI store only; don't conflate them.

Senior Engineer Tips

- Treat the keystore/certs like production credentials: base64 into encrypted CI secrets, decode to a temp file per build, clean up, and never let them touch the repo or logs — a leaked

keystore is a publish-as-you incident.

- Enroll in Play App Signing and use Fastlane `match` for iOS from the start; both eliminate the fragile, manual signing that breaks CI at the worst time (and `match` makes team signing reproducible).
- Set up dev/staging/prod flavors with distinct bundle ids and inject endpoints via `--dart-define`; hardcoded environments and single-id apps are recurring release/config bugs.

Architect Perspective

Signing + flavors are the build-configuration layer CI/CD depends on: secure secret handling turns sensitive keys into safely-injected temp files, and flavors give coexisting, environment-correct builds — with Fastlane automating the fiddly signing/store steps. Done right (encrypted secrets, Play App Signing/match, `--dart-define` config, per-flavor separation), it lets CI produce shippable, correct artifacts repeatably and securely — the bridge from a green build to an actual release (04_cd_release_automation.md, Module 51, Module 37).

Summary

- Sign releases: Android keystore (Play App Signing/upload key), iOS certs + provisioning (Fastlane `match`), on a Mac runner for iOS; secrets in encrypted CI store → temp files at build, never committed/logged.
- Flavors (dev/staging/prod): distinct bundle ids + name/icon + endpoint/config + Firebase config; inject env via `--flavor` + `--dart-define`/entry point (not hardcoded).
- Automate build+sign+upload with Fastlane lanes (or Codemagic built-ins); separate non-secret config from secrets.

Revision Notes

- Android: keystore (private key) + `signingConfigs` from git-ignored `key.properties`; Play App Signing (upload key). iOS: cert + provisioning profile (Apple Dev), Fastlane `match`/`sigh`, Mac runner.
- CI secrets: base64 keystore/`.p12`/profiles + passwords in encrypted store → decode to temp at build → clean up; never in repo/logs.
- Flavors: `productFlavors` (Android)/schemes+xcconfig (iOS) → distinct bundle id + name/icon + endpoint + Firebase config; Flutter `--flavor` + `--dart-define`/entry point (config, not secrets). Fastlane lanes (`match`/`gym`/`supply`/`pilot`) automate build+sign+upload.

Practice Questions

1. How do you get a signing keystore into CI without committing it?

2. What differs between dev/staging/prod flavors, and how is env config injected?
3. What does Fastlane automate, and why use `match`/Play App Signing?

Coding Questions

1. Configure Android signing from a git-ignored `key.properties` + a release `signingConfig`.
2. Define dev/staging/prod `productFlavors` (distinct ids) and a `--dart-define-from-file` build command.
3. Write a CI step decoding a base64 keystore secret and building a signed, flavored bundle.

Mini Project

Signing + flavors (Flutter/CI-CD): Configure Android signing (git-ignored `key.properties` + release `signingConfig`, Play-App-Signing-ready), dev/staging/prod flavors (distinct bundle ids + names + per-flavor endpoint via `--dart-define-from-file`), and a CI step that decodes a base64 keystore from an encrypted secret to build a signed prod bundle — plus a Fastlane lane sketch for iOS (`match` + `gym` + `pilot`). Acceptance: signing from git-ignored/encrypted secrets (never committed/logged); Play App Signing considered; dev/staging/prod flavors with distinct ids + injected (not hardcoded) endpoints; CI decodes keystore securely + builds signed flavored artifact; Fastlane lane for iOS; config vs secrets separated.

CD & Release Automation

> Continuous Delivery/Deployment automates the last mile: on a **tag/merge**, CI's signed artifact is **uploaded to a store track** — **Play Console tracks** (internal → closed/alpha → open/beta → production) or **App Store Connect / TestFlight** — via **Fastlane** (`supply` / `pilot`) or a mobile CI (Codemagic/Bitrise) built-in, with **automated version/build-number bumping**, **release notes/changelog**, and **staged (phased) rollout** (release to 1% → 100% of users, monitored) so a bad build affects few. Mobile is almost always **Continuous Delivery** (auto to beta, **manual/gated** promotion to production due to **store review** and business timing).

Introduction

This file covers the CD half: uploading to store tracks, version/build-number management, release notes, staged rollout, and the mobile reality that production release is gated (review + timing). It completes the pipeline started by CI + signing (02_ci_pipeline_and_automation.md/03_build_signing_and_flavors.md) and precedes deployment specifics (Module 51).

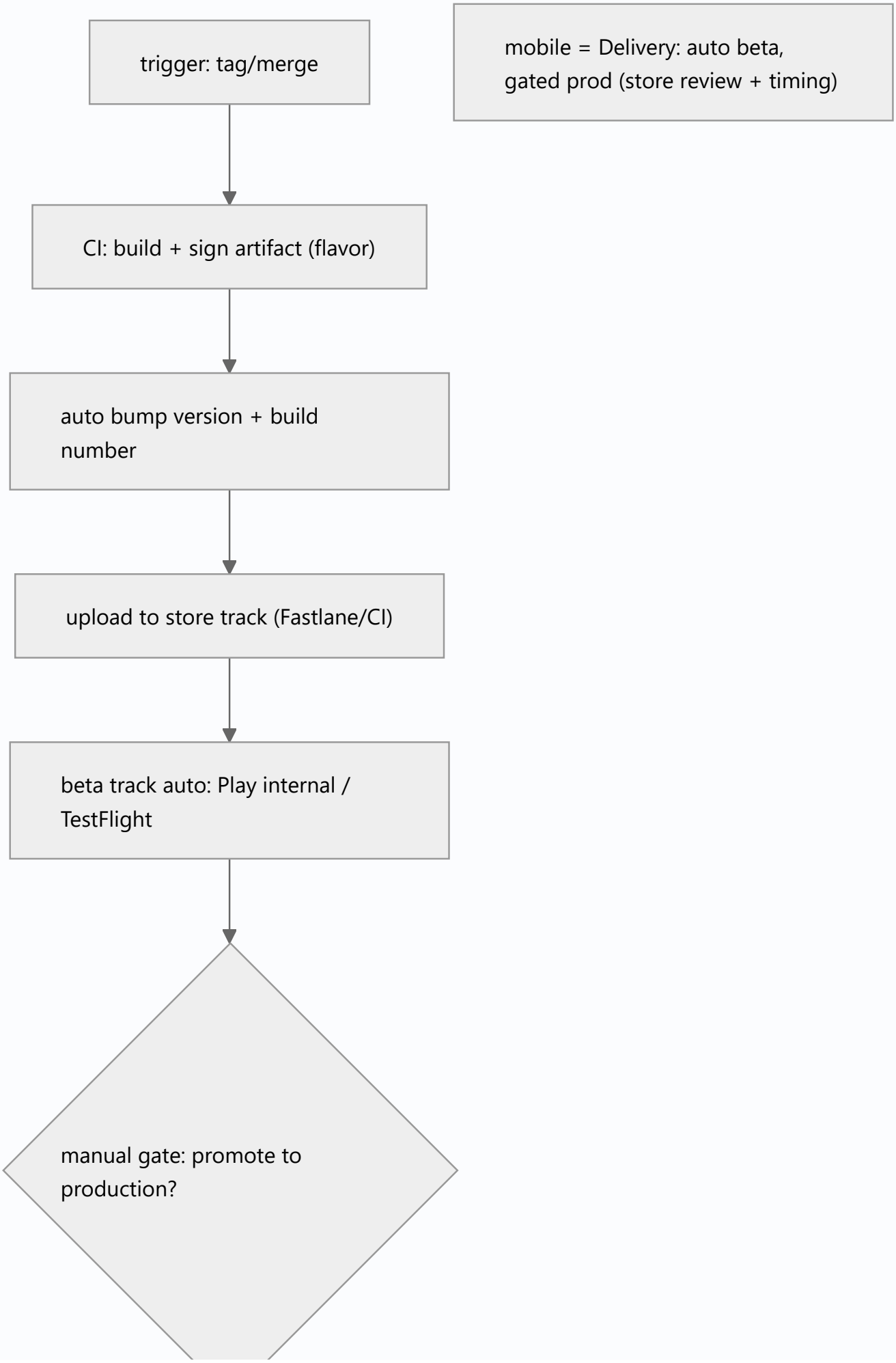
Why this concept exists

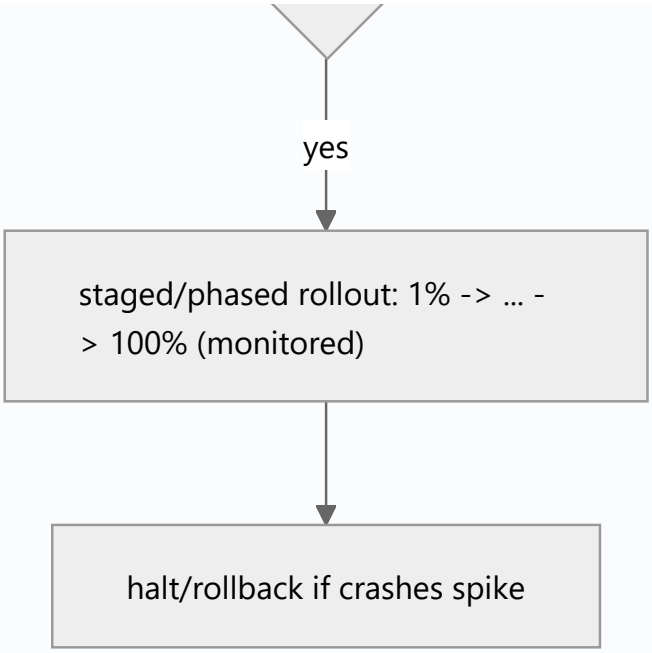
Manual store uploads are slow, error-prone rituals (wrong version, missing notes, forgot a track). Automating release — upload, versioning, notes, staged rollout — makes shipping **routine, repeatable, and safe**, and staged rollout **contains blast radius** so a regression hits 1% not 100%. Mobile's store-review gate makes full Continuous Deployment rare, so the automation targets **beta tracks automatically + a gated production promotion**.

Real-world analogy

CD is the **automated shipping + phased store rollout**: finished, sealed products (signed artifacts) are automatically sent to the **test shelf** (internal/beta) for staff/testers; a manager then **approves the retail launch**, which the system rolls out **gradually to stores** (1% of regions first, watching for complaints, then wider). Each shipment gets an **auto-incremented lot number** (version/build number) and a **what's-new note**. If early customers report a defect, you **halt the rollout** before it reaches everyone.

Internal Working


trigger: tag/merge
mobile = Delivery: auto beta, gated prod (store review + timing)
CI: build + sign artifact (flavor)
auto bump version + build number
upload to store track (Fastlane/CI)
beta track auto: Play internal / TestFlight
manual gate: promote to production?



- **Store tracks (progressive exposure)**:
  - **Play Console**: **internal** (fastest, small team) → **closed/alpha** → **open/beta** → **production**. Promote up the tracks as confidence grows.
  - **App Store Connect**: **TestFlight** (internal + external beta testers) → **App Store** (production, after review).
  - CD **auto-uploads to beta** (internal/TestFlight); **production is a gated promotion**.
- **Upload automation**:
  - **Fastlane**: `supply` (Play upload/promote), `pilot` (TestFlight), `deliver` (App Store metadata/submit).
  - **Codemagic/Bitrise**: built-in publishing to Play/App Store tracks (less scripting).
  - Needs **store API credentials** (Play service-account JSON, App Store Connect API key) in **encrypted secrets** (Module 37).
- **Versioning (automate it)**:
  - `version` **in** `pubspec.yaml` → `versionName+versionCode` (Android) / `CFBundleShortVersionString` + `CFBundleVersion` (iOS): a **user-facing version** (semver `1.2.0`) + a **monotonically increasing build number** (stores **reject duplicate build numbers**).
  - Automate the **build-number bump** (e.g., from CI run number / git commit count / date) so every upload is unique; bump the **semver version** per release (conventional commits/tags). Never upload a duplicate build number.
- **Release notes / changelog**: generate from **conventional commits**/PR titles or a maintained `CHANGELOG`; supply per-track/locale notes (Fastlane `supply` / `deliver` metadata). "What's new" is required for store releases.
- **Staged / phased rollout (containment)**:

    - **Play**: **staged rollout %** (e.g., 5% → 20% → 50% → 100%) — bad build reaches few; **halt** or **roll back** if metrics degrade.
    - **App Store**: **Phased Release** (automatic 7-day gradual rollout for auto-updates) + ability to pause.
    - Pair with **monitoring** (crash-free rate, errors — Module 52) to decide promote/halt.
- **The manual gate (mobile reality)**: production release is usually **manual/approved** (Continuous Delivery) because of **store review latency**, **release timing/marketing**, and **rollback difficulty on mobile** (users must update). CD automates everything **up to** that gate; the gate is a **one-click promotion**, not a manual build.
- **Rollback reality**: mobile can't instantly roll back a deployed build (users have it) — mitigations: **staged rollout + halt**, **server-side feature flags/kill switches**, **hotfix + expedited release**. Design for **forward-fix**, not easy rollback.
- **Release triggers**: **tag** (`v1.2.0`) → production pipeline; **merge to main** → beta; keeps release intentional + traceable.

Memory Representation

Not runtime — a **release pipeline** (build → version → upload → track → rollout) + release metadata (version, build number, notes, rollout %). Store-side: tracks with staged-rollout state; monitoring feeds promote/halt decisions.

Compiler / Build Behavior

CD consumes the signed artifact from CI (03_build_signing_and_flavors.md); version/build number are set at build (via `--build-name` / `--build-number` or pubspec) and embedded in the artifact.

Runtime Behavior

Beta uploads are automatic; production is gated then rolled out gradually; a bad release can be **halted** at a low % before wide exposure. Users receive the update progressively (auto-update/phased).

Flutter Engine Behavior

Not applicable (store distribution). The shipped artifact runs normally per flavor.

Dart VM Behavior

Not applicable.

Examples

```
# Fastlane — auto-bump build number, upload to beta, staged production promote
lane :beta do
  build_number = number_of_commits            # unique, monotonic
  build_app(scheme: "prod")
  upload_to_testflight(build_number: build_number)      # iOS beta
  # Android beta: supply(track: 'internal', aab: '...')
end
lane :production do
  supply(track: 'production', rollout: '0.05')  # Play staged rollout 5%
  # promote later: supply(track: 'production', rollout: '1.0') after monitoring
end
```

```
# CI release job (on tag) — build signed, bump build number, deploy to beta
on: { push: { tags: ['v*'] } }
steps:
  - run: flutter build appbundle --release --flavor prod \
         --build-name=${{ github.ref_name }} --build-number=${{ github.run_number }}
  - run: bundle exec fastlane beta         # upload to Play internal / TestFlight
# production promotion = manual approval step (environment protection) -> staged rollout
```

```
Release flow (Continuous Delivery, mobile):
  tag v1.2.0 -> CI build+sign -> auto build-number bump -> upload BETA (internal/TestFlight)
  -> [MANUAL one-click promote] -> PRODUCTION staged rollout 5% -> monitor -> 20% -> ... -> 100%
  (halt/rollback via rollout pause + feature flags if crash-free rate drops)
```

Diagrams

Common Mistakes

Mistake	Why it's a problem	Fix
Duplicate build number	Store rejects upload	Auto-bump unique monotonic build number
Full 100% release immediately	Bad build hits everyone	Staged/phased rollout + monitor
Manual store uploads	Slow/error-prone ritual	Automate (Fastlane/CI publishing)
Auto-deploy straight to prod (mobile)	Ignores review/timing/rollback	Continuous Delivery: gated prod promotion
No release notes/changelog	Store requires "what's new"	Generate from commits/CHANGELOG
Expecting easy rollback	Users already have the build	Staged rollout + flags + forward-fix
Store credentials in repo	Security breach	Encrypted secrets (service account/API key)
No monitoring during rollout	Can't detect a bad release	Watch crash-free rate; halt if degraded

Best Practices

- Automate **upload to beta tracks** (Play internal / TestFlight) on merge/tag via **Fastlane (`supply` / `pilot`)** or CI built-ins; **gate production** as a **one-click promotion** (Continuous Delivery — store review/timing).
- **Auto-bump a unique monotonic build number** (CI run/commit count) + semver `version`; **never upload a duplicate build number**; generate **release notes** from commits/CHANGELOG.
- Use **staged/phased rollout** (5% → 100%) with **monitoring** (crash-free rate — Module 52) and **halt** on degradation; design for **forward-fix + feature flags/kill switches** (mobile rollback is hard).
- Store **store API credentials in encrypted secrets**; trigger production releases from **tags** for traceability.

Performance

Not runtime perf; the concern is **release safety + speed**: automation removes slow manual steps, and staged rollout **contains blast radius** (a bad build monitored at 5% beats a 100% incident). Monitoring-driven promote/halt is the safety mechanism (Module 52).

Advantages / Disadvantages

- **+** Fast, repeatable releases; automatic beta distribution; safe production via staged rollout + monitoring; auto versioning/notes; traceable (tag-triggered).
- **−** Mobile can't easily roll back (forward-fix), store-review latency/gate, store-credential + metadata setup, staged-rollout monitoring discipline.

Interview Questions

1. 🟢 **What are the store tracks, and how does CD use them?** — Play: internal→closed/alpha→open/beta→production; App Store: TestFlight→App Store. CD auto-uploads to beta; production is a gated promotion.
2. 🟢 **Why is mobile usually Continuous Delivery, not Deployment?** — Store review latency, release timing, and hard rollback make production a gated (manual one-click) promotion rather than fully automatic.
3. 🟡 **Why must the build number auto-increment, and how?** — Stores reject duplicate build numbers; derive it from CI run number/commit count/date so each upload is unique + monotonic.
4. 🟡 **What is staged/phased rollout and why use it?** — Releasing to a growing % of users (5%→100%) while monitoring, so a bad build reaches few and can be halted — blast-radius containment.
5. 🟡 **How do you automate store uploads?** — Fastlane (`supply` for Play, `pilot` / `deliver` for App Store) or Codemagic/Bitrise built-ins, using store API credentials from encrypted secrets.
6. 🔴 **How do you handle a bad release given mobile can't easily roll back?** — Staged rollout + halt, server-side feature flags/kill switches, and a forward-fix (hotfix + expedited release) — plan for forward-fix, not rollback.
7. 🔴 **What triggers a production release, and why?** — A version tag (`v1.2.0`) → the production pipeline (after a manual promote), keeping releases intentional + traceable.

Senior Engineer Tips

- Automate to beta, gate to production: wire tag→beta automatically and make prod a one-click promote with a staged rollout; that's the safe, standard mobile CD shape.
- Auto-bump a unique build number from the CI run/commit count and never think about it again; duplicate-build-number rejections are a needless recurring release failure.
- Always release staged + monitored and design for forward-fix (flags/kill switches + hotfix path); assuming you can "just roll back" a mobile release is how a 5% issue becomes a 100% incident.

Architect Perspective

CD/release automation is the last mile that makes shipping routine and safe: signed artifacts flow automatically to beta with auto-versioning/notes, production is a gated one-click promotion, and staged rollout + monitoring contain risk in a world where rollback is hard. Embracing Continuous Delivery (not Deployment) for mobile, with forward-fix and feature-flag strategies, turns releases

from fragile rituals into a controlled, observable process — the culmination of the CI/signing pipeline and the handoff to deployment + monitoring (02_ci_pipeline_and_automation.md, Module 51, Module 52).

Summary

- CD auto-uploads signed artifacts to beta tracks (Play internal/TestFlight) via Fastlane/CI, with auto build-number bump + release notes; production is a gated one-click promotion (mobile = Continuous Delivery).
- Use staged/phased rollout (5%→100%) with monitoring + halt; design for forward-fix + feature flags/kill switches (mobile rollback is hard).
- Store credentials in encrypted secrets; trigger production from tags for traceability.

Revision Notes

- Tracks: Play internal→closed/alpha→open/beta→production; App Store TestFlight→production. CD: auto beta, gated (manual one-click) prod (store review/timing) = Continuous Delivery.
- Upload: Fastlane `supply` (Play)/ `pilot` + `deliver` (App Store) or Codemagic/Bitrise built-in; store API creds (Play service-account JSON / ASC API key) in encrypted secrets.
- Versioning: pubspec `version` → versionName+versionCode/CFBundle*; auto-bump unique monotonic build number (CI run/commit count); never duplicate. Release notes from commits/CHANGELOG.
- Staged rollout 5%→100% + monitoring (crash-free) + halt; mobile rollback hard → forward-fix + feature flags/kill switches; production triggered by tag.

Practice Questions

1. Why is production release gated (Delivery) on mobile?
2. How and why do you auto-increment build numbers?
3. How do you contain the risk of a bad release given hard rollback?

Coding Questions

1. Write a Fastlane lane uploading to a beta track with an auto build number.
2. Configure a staged production rollout (e.g., Play 5%) + promotion step.
3. Add a CI release job (tag-triggered) that builds signed + deploys to beta with a manual prod gate.

Mini Project

Release automation (Flutter/CI-CD): Design a CD flow: on tag, CI builds a signed prod artifact with an auto-bumped unique build number + generated release notes, uploads to beta (Play internal/TestFlight) automatically, and gates production behind a manual one-click promotion that does a **staged rollout** (5%→100%) with crash-free-rate monitoring + halt — using Fastlane (`supply` / `pilot`) and store credentials from encrypted secrets. Note the forward-fix/feature-flag rollback strategy. Acceptance: auto beta upload + gated prod (Continuous Delivery); unique monotonic build number + release notes; staged rollout + monitoring + halt; credentials in encrypted secrets; forward-fix/flags rollback plan; tag-triggered + traceable.

CI/CD Integration (Capstone: Commit → Store Pipeline)

> Assemble the full pipeline from **commit to store**: on **PR**, run fast **analyze + unit/widget tests** (cached, parallel, incremental via `melos --since`) and **gate the merge**; on **merge to main**, build **signed** artifacts per flavor and **auto-deploy to beta** (Play internal / TestFlight) with an **auto-bumped build number**; on **tag**, do a **gated one-click promotion** to production with **staged rollout + monitoring**. All secrets (keystore/certs/store credentials) live in the **encrypted CI store**; the whole thing is **config-as-code**, reproducible, and documented. This is the delivery backbone that makes testing enforceable and releases routine.

Introduction

This module capstone composes fundamentals, the CI pipeline, signing/flavors, and CD/release automation into one coherent commit→store pipeline — the reference a team runs. It shows the stages, triggers, gates, secrets, and the branching/versioning strategy end to end.

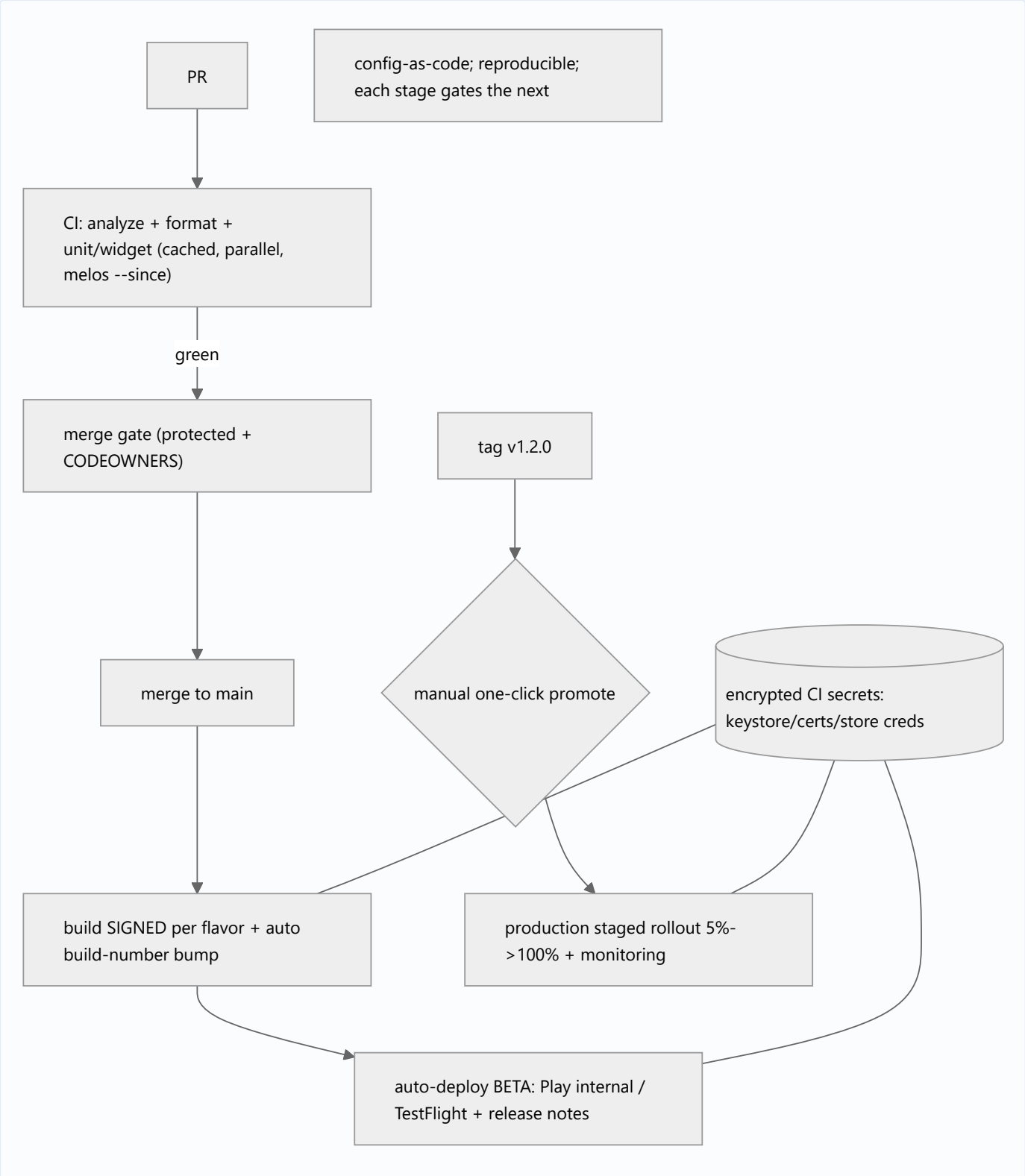
Why this concept exists

The pieces (CI, signing, CD) only deliver value **assembled into one gated, automated, reproducible pipeline** with the right triggers and secret handling. This capstone provides that assembled exemplar — turning "we have some CI" into "commit → tested → signed → beta → gated staged production," the standard for professional Flutter delivery.

Real-world analogy

It's the **complete factory-to-retail line**: inspection at every added part (PR CI), assembly + sealing + shipping to the **test shelf** on completion (merge → signed → beta), and a **manager-approved, phased retail launch** with monitoring (tag → gated staged production). One integrated, automated line — not disconnected manual steps — with the **vault** (encrypted secrets) securing the seals.

Internal Working



- **PR trigger — fast CI + gate** (02_ci_pipeline_and_automation.md): pinned SDK → cached `pub get` → `dart format` + `flutter analyze` → `flutter test --coverage` (unit/widget), parallel + `melos --since` incremental (monorepo). **Merge gated** on green (protected branch + CODEOWNERS — Module 47). E2E on nightly/merge (not every PR).
- **Merge trigger — build + beta**: build **signed** artifacts per **flavor** (03_build_signing_and_flavors.md) with an **auto-bumped unique build number**; **auto-**

deploy to beta (Play internal / TestFlight) with **generated release notes** (04_cd_release_automation.md). This is Continuous Delivery to staging.

- **Tag trigger — gated production**: a **version tag** (`v1.2.0`) → a **manual one-click promotion** (environment protection/approval) → **staged rollout (5%→100%)** with **crash-free-rate monitoring** + **halt** (Module 52). Mobile = Continuous **Delivery** (gated prod).
- **Secrets everywhere-secure** (Module 37): keystore/certs/store-API credentials in the **CI encrypted store**, decoded to temp files at build, never committed/logged.
- **Config-as-code + reproducible**: the whole pipeline is versioned YAML/config with **pinned toolchains** and clean runners; iOS builds on **Mac runners**; Fastlane (or Codemagic/Bitrise built-ins) automate build/sign/upload.
- **Branching/versioning strategy** (glue it together): trunk-based or GitFlow-lite — **PRs → main (gated)**, **main → beta (auto)**, **tags → production (gated, staged)**; semver bumps per release, build number auto per upload.
- **The payoff**: testing is **enforced** (gate), builds are **signed + reproducible**, releases are **automated + staged + monitored** — commit to store is a controlled, fast, safe path enabling confident, frequent shipping.
- **Right-sizing**: a solo/small app needs the CI half + simple beta deploy; large/multi-team adds incremental monorepo CI, flavors, staged rollout, and strict gating — scale the pipeline to the app (Module 47).

Memory Representation

Not runtime — a **pipeline definition** (jobs/triggers/gates/secrets) + release metadata (version/build number/notes/rollout state) + run history. The invariant: main always green + buildable; each stage gates the next; secrets never leave the encrypted store.

Compiler / Build Behavior

Pinned SDK + clean runners → reproducible builds; per-flavor signed artifacts; version/build number embedded; caching speeds compilation. Undeclared secrets or duplicate build numbers fail the build/upload.

Runtime Behavior

PR → fast verify + gate; merge → signed build + auto beta; tag → gated staged production with monitoring-driven promote/halt. Failures fail fast; users receive production progressively (staged).

Flutter Engine Behavior

Integration-test jobs run the engine on emulators; production artifacts run per flavor on-device. Otherwise pipeline stages build/test, not run, the engine.

Dart VM Behavior

Fast unit base + incremental (`melos --since`) keep CI quick; tests run in the VM; builds AOT-compile per flavor.

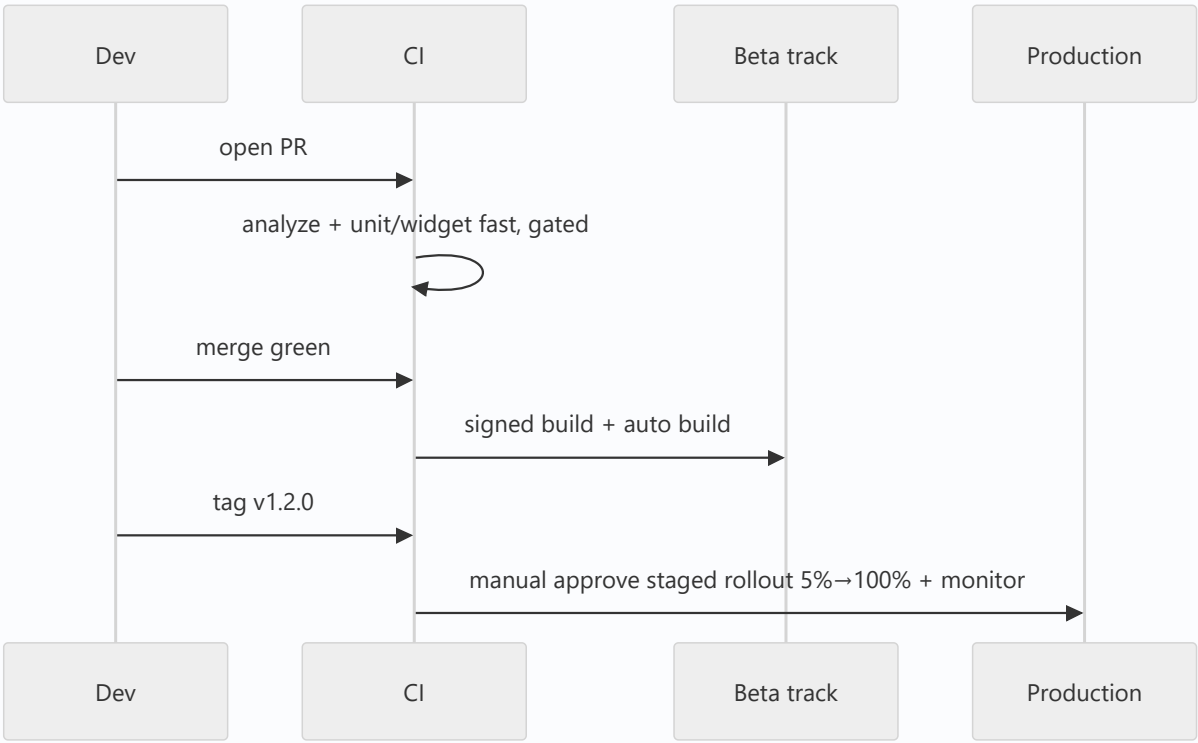
Examples

```yaml
# Full pipeline (GitHub Actions sketch), triggers → stages
name: CI-CD
on:
  pull_request: {}                # fast CI + gate
  push: { branches: [main] }      # -> beta
  push: { tags: ['v*'] }          # -> gated production

jobs:
  verify:                         # PR + push: fast, cached, gated
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: subosito/flutter-action@v2
        with: { flutter-version: '3.24.0', cache: true }
      - run: flutter pub get
      - run: dart format --set-exit-if-changed . && flutter analyze
      - run: flutter test --coverage          # (or: melos run test --since=origin/main)

  beta:                           # on merge to main: signed build -> beta
    if: github.ref == 'refs/heads/main'
    needs: verify
    runs-on: macos-latest         # iOS needs macOS
    steps:
      - run: echo "$KEYSTORE_B64" | base64 -d > android/app/upload.jks   # from encrypted secret
      - run: flutter build appbundle --release --flavor prod \
             --build-number=${{ github.run_number }}                    # unique build number
      - run: bundle exec fastlane beta                                   # -> Play internal / TestFlight
```

```yaml
  production:                       # on tag: MANUAL gate -> staged rollout
    if: startsWith(github.ref, 'refs/tags/v')
    needs: verify
    environment: production         # requires manual approval
    runs-on: macos-latest
    steps:
      - run: bundle exec fastlane production   # supply(track: production, rollout: '0.05') + monitor
```

```
Trigger -> stage map:
  PR            -> analyze + format + unit/widget (cached, gated)   [fast feedback]
  merge->main   -> signed build (flavor) + auto build# + BETA deploy [Continuous Delivery]
  tag v*        -> manual promote -> PROD staged rollout + monitoring [gated]
  nightly       -> integration/E2E on emulators
```

Diagrams



Common Mistakes

Mistake	Why	Fix
No merge gate	Bad code on main	Protected branch requires green CI
Auto-deploy straight to prod (mobile)	Ignores review/timing/rollback	Continuous Delivery: gated prod promotion
Slow PR pipeline (E2E/full builds)	Devs bypass CI	Fast PR (analyze+unit/widget); heavy on merge/tag/nightly
Secrets in repo/config/logs	Security breach	Encrypted CI secrets → temp → cleanup
Duplicate build number	Store rejects	Auto-bump unique build number
100% release immediately	Bad build → everyone	Staged rollout + monitoring + halt
Non-reproducible builds	"works on my machine"	Pin SDK, clean runners
One-size pipeline for a tiny app	Over-engineering	Right-size (CI + simple beta first)

Best Practices

- **Config-as-code** pipeline with **pinned SDK/clean runners**: **PR → fast gated CI** (analyze/format/unit/widget, cached/parallel/ `melos --since`), **merge → signed per-flavor build + auto beta** (unique build number + notes), **tag → gated one-click prod promotion + staged rollout + monitoring**.
- Keep **all secrets** (keystore/certs/store creds) in the **encrypted CI store** (temp files, cleaned up); build iOS on **Mac runners**; automate build/sign/upload with **Fastlane/Codemagic**.
- **Gate merges** (protected + CODEOWNERS + coverage) and **gate production** (manual approval); design for **forward-fix + feature flags** (hard mobile rollback).
- Define a clear **branching/versioning strategy** (PR→main→beta→tag→staged prod); **right-size** the pipeline to the app's stage.

Performance

Caching + parallelism + `melos --since` keep PR feedback fast; heavy stages deferred to merge/tag/nightly keep the inner loop tight; staged rollout limits release blast radius. A fast, reliable pipeline is trusted; a slow/flaky one is bypassed (Module 47).

Advantages / Disadvantages

- **+** Enforced testing, reproducible signed builds, automated + staged + monitored releases, fast trusted feedback, routine safe shipping — enables scale.
- **−** Upfront + ongoing setup (pipelines/secrets/signing/store), CI cost + Mac runners for iOS, discipline (gates, small merges), mobile prod-gate/rollback nuances.

Interview Questions

1. 🟢 **Describe a full commit→store pipeline.** — PR: fast gated CI (analyze/unit/widget); merge→main: signed per-flavor build + auto beta (unique build number + notes); tag: manual-gated production promotion + staged rollout + monitoring.
2. 🟢 **What runs on PR vs merge vs tag?** — PR: analyze + unit/widget (fast, gated); merge: signed build + beta deploy; tag: gated staged production; nightly: E2E.
3. 🟡 **How are secrets handled across the pipeline?** — In the CI encrypted store (keystore/certs/store creds), decoded to temp files at build, never committed/logged; iOS on Mac runners.
4. 🟡 **Why is production gated and staged for mobile?** — Store review/timing + hard rollback → a manual one-click promotion + staged rollout with monitoring (Continuous Delivery), containing risk.
5. 🟡 **How do you keep the pipeline fast and reproducible?** — Cache + parallelize + `melos --since` incremental, pin the SDK, clean runners; keep heavy stages off the PR path.
6. 🔴 **How do you handle a bad production release?** — Halt the staged rollout, use feature flags/kill switches, and ship a forward-fix (hotfix + expedited release) — mobile can't easily roll back.
7. 🔴 **How do you right-size the pipeline?** — Small app: CI + simple beta deploy; large/multi-team: add incremental monorepo CI, flavors, staged rollout, strict gating — scale to the app's stage.

Senior Engineer Tips

- Build the pipeline in the order commit→beta→gated-prod, get the PR feedback fast and gated first, then add signed beta, then the gated staged production; each layer compounds and stays trusted.
- Treat secrets and reproducibility as non-negotiable from day one (encrypted store + pinned SDK); leaked keystores and non-deterministic builds are the incidents that hurt most.
- Make production a one-click gated staged rollout with monitoring and a forward-fix plan; auto-to-prod on mobile trades a rare convenience for a real risk you can't easily undo.

Architect Perspective

The commit→store pipeline is the delivery backbone that operationalizes everything upstream: testing becomes enforced (gate), builds become signed + reproducible (config-as-code + secure secrets), and releases become automated, staged, and monitored (Continuous Delivery with a gated prod). Assembled and right-sized, it turns shipping from a fragile ritual into a fast, safe, routine flow — the enabler of scalable, multi-team Flutter delivery and the handoff to deployment + monitoring (Module 49, Module 51, Module 52, Module 47).

Summary

- Full pipeline: PR → fast gated CI (analyze/unit/widget, cached/parallel/incremental); merge → signed per-flavor build + auto beta (unique build number + notes); tag → gated one-click prod promotion + staged rollout + monitoring.
- Secrets in the encrypted CI store (temp/cleanup); config-as-code + pinned SDK (reproducible); Fastlane/Codemagic automate build/sign/upload; iOS on Mac runners.
- Mobile = Continuous Delivery (gated prod, staged, forward-fix); right-size to the app's stage; the backbone that makes testing enforced + releases routine.

Revision Notes

- Triggers→stages: PR (analyze/format/unit/widget, cached/parallel/ `melos --since` , merge-gated) → merge/main (signed per-flavor build + auto build# + beta deploy + notes) → tag (manual gate → prod staged rollout 5%→100% + monitoring) → nightly (E2E).
- Secrets in encrypted CI store (keystore/certs/store creds → temp → cleanup); config-as-code + pinned SDK/clean runners (reproducible); Mac runner for iOS; Fastlane/Codemagic automate build/sign/upload.
- Mobile = Continuous Delivery (gated prod), staged rollout + halt + forward-fix/flags; branching: PR→main→beta→tag→staged prod; semver + auto build#; right-size to stage.

Practice Questions

1. Map triggers (PR/merge/tag/nightly) to pipeline stages.
2. How are secrets, signing, and reproducibility handled across the pipeline?
3. Why is mobile production gated + staged, and how do you handle a bad release?

Coding Questions

1. Write a full CI/CD config with PR (gated CI), merge (signed beta), and tag (gated staged prod) jobs.
2. Add secure keystore decoding + auto build-number bump.
3. Configure a manual-approval production job with a staged rollout.

Mini Project

Commit→store pipeline (capstone — Flutter/CI-CD): Author a full GitHub Actions (or Codemagic) CI/CD pipeline: PR job (pinned SDK, cached, `format` + `analyze` + `flutter test --coverage` , merge-gated; `melos --since` if monorepo), merge job (signed per-flavor build with auto build-number bump → auto-deploy beta via Fastlane + release notes), and tag job (manual-approval

production → staged rollout 5%→100% with monitoring) — all secrets in the encrypted store, iOS on a Mac runner. Document the branching/versioning strategy + forward-fix rollback plan. Acceptance: config-as-code, pinned SDK; PR fast + merge-gated; merge→signed beta (unique build#, notes); tag→gated staged prod + monitoring; secrets encrypted (temp/cleanup, never committed); Fastlane/Codemagic automation; branching/versioning + forward-fix documented; right-sized.