60 · Software Engineering Practices

Flutter Practice Notes

Contents

60 · Software Engineering Practices

> The craft *around* the code — how professional engineers version, review, document, ship, and observe software as a team — the difference between "can code" and "is an engineer."

Why this module exists

You can be excellent at Dart and Flutter and still fail on a real team, because ~half of senior engineering is not writing code — it is the **practices** that let many people change one codebase safely, repeatedly, for years:

- A feature works on your machine but the **PR** is unreviewable, so it sits for three days.
- The **release** breaks prod because there was no flag to turn it off and no way to roll back.
- An incident happens and nobody can answer *"is it broken, for whom, since when?"* because there is no **observability**.
- A decision made in a hallway is re-litigated every quarter because it was never written into a **design doc**.

These are not Flutter problems — they are **engineering** problems, and every product company interviews for them ("tell me about your code review process," "how do you do releases," "walk me through an incident"). This module teaches that layer, vendor- and framework-neutral.

Where a topic already has a tool-level module in this handbook (CI/CD, Monitoring, Logging, Deployment), this module teaches the **discipline and decision-making** — *why* you gate a merge, *how* you choose a release strategy, *what* makes a system observable — and cross-links to the mechanics rather than repeating them.

What you will be able to do

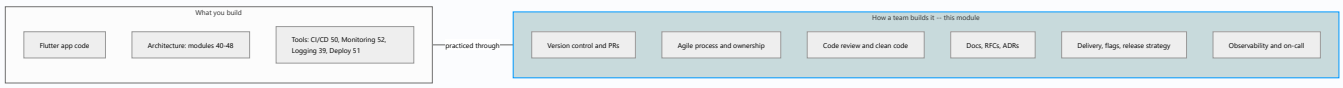
- Use **Git** with intent: branching model, atomic commits, rebase vs merge, resolving conflicts, and a clean PR — plus a mental model of Git's object store.
- Run inside an **Agile** team: sprints, estimation, stand-ups, retros — and collaborate through **code ownership** and clear boundaries.
- Give and receive **code reviews** that improve the code without stalling it, and apply **clean-code / refactoring** discipline continuously.
- Write the documents engineers actually rely on: **READMEs, RFCs / design docs, and ADRs** — and know which to reach for.
- Ship safely: **CI/CD as a practice**, **feature flags**, **release strategies** (canary, blue-green, staged rollout), and **App Store / Play Store** publishing.

- Make a system **observable** — logs, metrics, traces — and reason about SLIs/SLOs, alerting, and on-call as a discipline.

File index

#	File	Practice focus	Pairs with (mechanics)
1	`01_version_control_git.md`	Git object model, branching, PRs, rebase vs merge, GitHub/GitLab flow, monorepo vs polyrepo	45 Modular · monorepo & melos
2	`02_agile_and_collaboration.md`	Agile/Scrum/Kanban, estimation, ceremonies, code ownership, working in a team	58 Senior Architect Notes · leadership
3	`03_code_quality_and_review.md`	Clean code, refactoring, code review, linters, engineering best practices	04 SOLID, 49 Testing
4	`04_documentation_and_design_docs.md`	READMEs, RFCs / design docs, ADRs, diagrams, writing for engineers	58 · decision frameworks & ADRs
5	`05_delivery_and_release.md`	CI/CD as practice, feature flags, canary/blue-green/staged rollout, store publishing	50 CI CD, 51 Deployment
6	`06_observability_as_practice.md`	Logs/metrics/traces, SLI/SLO/SLA, alerting, on-call, incident response	52 Monitoring, 39 Logging

How this module relates to the rest of the handbook



> The other modules give you the *artifact and the tools*. This module gives you the *practices that turn a group of coders into an engineering team*.

Summary

- Half of senior engineering is the craft around the code: version control, process, review, docs, delivery, observability.
- Six topic files, each teaching the discipline vendor-neutrally and cross-linking to the tool-level module where one exists.
- These are exactly the behaviors product companies interview for beyond coding ability.

Revision Notes

- "Works on my machine" is not done — reviewable PR, safe release, and observability are part of done.
- Git object model → branching → PR; Agile ceremonies → ownership; review + clean code; RFC/ADR docs; flags + release strategy; logs/metrics/traces + SLOs.
- Where a tool module exists (50/51/52/39), this module owns the *why and how-as-a-team*; the tool module owns the *mechanics*.

Version Control with Git

> Git is a distributed, content-addressable filesystem that models a project's history as an immutable directed acyclic graph of snapshots, with branches and tags as cheap movable pointers into that graph.

Introduction

Version control is the discipline of recording *how* a codebase changes over time so that any past state can be recovered, any change can be attributed, and many people can work in parallel without overwriting each other. Git is the de facto standard. What makes Git worth understanding at the internals level is that it is not a "diff database" the way older systems (SVN, CVS) were — it is a small key-value store of immutable objects on top of which every command is a graph operation. Once you see the object model, commands like `rebase`, `reset`, and `cherry-pick` stop being magic incantations and become obvious manipulations of pointers and snapshots.

This handbook is a *practice* topic: it teaches how a professional team actually uses Git day to day, but it grounds every workflow in what Git is doing to its objects and refs underneath.

Why this concept exists

Before version control, teams shared code by copying folders (`project_final_v2_REALFINAL/`) and emailing zip files. That approach fails on every axis that matters:

- **No history**: you cannot see what changed, when, or why.
- **No attribution**: you cannot blame a regression on a specific change.
- **No parallelism**: two people editing the same file overwrite each other.
- **No safe experimentation**: trying a risky idea means risking the whole project.
- **No auditable release**: you cannot say "production is exactly commit X."

Git exists to make history a *first-class, tamper-evident, distributed* data structure. Because each commit is addressed by the SHA-1/SHA-256 hash of its content (including its parent), history cannot be silently rewritten — any change to any ancestor changes every descendant hash. Because every clone is a full copy of the graph, there is no single point of failure and most operations are local and instant.

Real-world analogy

Think of Git as a **library of numbered photographs of your project**, plus a wall of sticky notes.

- Each **commit** is a full photograph (snapshot) of every file at a moment in time — not a list of edits, a whole picture.
- Photographs are filed by a fingerprint of their contents (the SHA), so identical photos are stored once and no photo can be altered without changing its fingerprint.
- Each photo has a note on the back saying "the photo before this one was #abc123" — that back-reference chains the photos into history.
- **Branches** and **HEAD** are sticky notes you stick onto whichever photo is "current." Moving a branch is just peeling the note off one photo and sticking it on another — the photos never move.

Merging is taking two lines of photographs that diverged and producing a new photo that reconciles both. Rebasing is *re-shooting* your photos as if you had started from a later point in the other line's history.

Problem Statement

A team of engineers must simultaneously:

1. Develop multiple features and fixes in parallel.
2. Keep an always-releasable mainline.
3. Recover any historical state and understand why each change was made.
4. Integrate work from many contributors with minimal collision.
5. Tie a deployed artifact to an exact, verifiable source state.

The core question is: *what data structure lets many people evolve a shared codebase independently and reconcile safely, while keeping history immutable and auditable?* Git's answer is a content-addressable object store shaped as a commit DAG, with mutable refs layered on top.

Internal Working

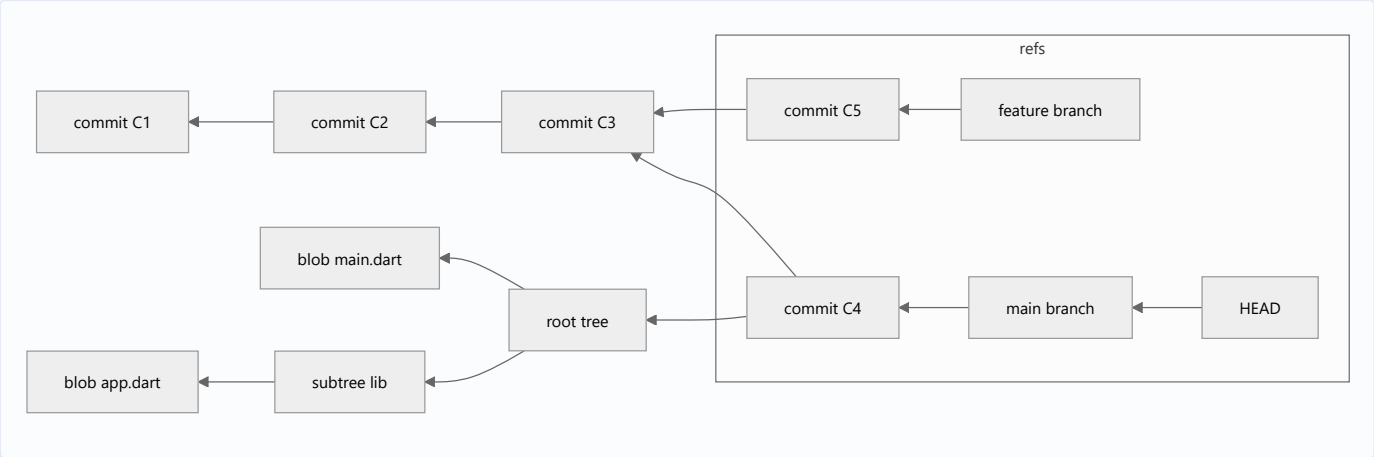
Git stores four object types, each addressed by the hash of its content:

- **blob** — the raw bytes of a file (no filename, no metadata).
- **tree** — a directory listing: names → (mode, blob hash | subtree hash). Trees give blobs their filenames and nest to form the directory structure.
- **commit** — a snapshot pointer: the root tree hash, zero or more parent commit hashes, author/committer, timestamp, and message.
- **tag** (annotated) — a named, signed pointer to a commit with its own message.

Refs are just files containing a hash. A branch (`refs/heads/main`) is a pointer to a commit. `HEAD` usually points to a branch (`ref: refs/heads/main`); when it points directly at a commit you are in "detached HEAD." The **index** (staging area) is a binary file listing the tree you are *about* to

commit.

Because each commit names its parents, commits form a **directed acyclic graph**. A merge commit simply has two (or more) parents.



The arrows point from child to parent because a commit knows its ancestors, never its descendants. Reachability from a ref is what keeps objects alive; unreachable objects are eventually garbage-collected.

Memory Representation

On disk, everything lives under `.git/`:

- `.git/objects/` — the object database. A **loose object** is stored at `objects/ab/cdef...` where `ab` is the first two hex digits of the SHA and the rest is the filename. The file is the object's content, zlib-compressed, prefixed with a header like `blob 128\0`.
- `.git/refs/heads/<branch>`, `.git/refs/tags/<tag>` — plain files containing a 40-char (SHA-1) hash. Often collapsed into `.git/packed-refs`.
- `.git/HEAD` — the symbolic ref for the current branch.
- `.git/index` — the staging area (a sorted list of path/blob/stat entries).
- `.git/config`, `.git/hooks/`, `.git/logs/` (the reflog).

As history grows, thousands of loose objects are inefficient, so `git gc` compresses them into **packfiles** (`.git/objects/pack/*.pack` + `.idx`). Packfiles store objects **delta-compressed** against similar objects — this is where "Git stores snapshots, not diffs" gets nuanced: the *model* is snapshots, but the *storage* uses deltas as a compression detail, transparent to the model.

Content addressing means deduplication is automatic: two identical files anywhere in history share one blob; an unchanged file across 1000 commits is one blob referenced by 1000 trees.

To inspect the raw store:

```
git cat-file -t <sha>     # object type: blob | tree | commit | tag
git cat-file -p <sha>     # pretty-print object content
git rev-parse HEAD        # resolve a ref to its SHA
```

Compiler Behavior

Not applicable — because Git is a version-control tool operating on files as opaque bytes; it never compiles or parses Dart. Compilation of your Flutter/Dart code is handled separately by the Dart toolchain and is unrelated to how Git stores or moves commits.

Runtime Behavior

Git commands are graph and pointer manipulations over the object store and refs:

- **commit** — writes blobs/trees for the staged content, writes a new commit object whose parent is the current `HEAD` commit, then advances the current branch ref to the new commit.
- **branch** — writes a new ref file pointing at a commit. O(1), no copying.
- **checkout/switch** — moves `HEAD` and rewrites the working tree + index to match the target commit's tree.
- **merge** — finds the merge base (common ancestor); if one side is an ancestor of the other, does a **fast-forward** (just moves the ref). Otherwise creates a new **merge commit** with two parents combining both trees.
- **rebase** — replays your commits one-by-one onto a new base, creating *new* commits with new SHAs (old ones become unreachable). Rewrites history.
- **cherry-pick** — applies the diff introduced by one commit as a new commit on the current branch.
- **reset** — moves the current branch ref to another commit. `--soft` keeps index+worktree; `--mixed` (default) resets index, keeps worktree; `--hard` resets both (destructive to uncommitted work).
- **revert** — creates a *new* commit that undoes a previous commit's changes; history is preserved (safe on shared branches).
- **stash** — saves worktree+index as special commits stored off to the side, then restores a clean worktree.

The **reflog** (`git reflog`) records every movement of `HEAD` and branch tips, so "lost" commits from a bad reset/rebase are usually recoverable for ~90 days.

Flutter Engine Behavior (if applicable)

Not applicable — because the Flutter engine renders and runs your app at runtime; it has no involvement in how source history is recorded. Git operates purely on repository files.

Dart VM Behavior (if applicable)

Not applicable — because the Dart VM executes Dart bytecode/AOT code; version control happens entirely at the source/tooling layer and never touches VM execution.

Examples

Create a feature branch, commit atomically, and push:

```
git switch -c feature/login-form        # create + switch to a branch
# ... edit files ...
git add lib/features/auth/login_form.dart
git commit -m "Add email/password login form widget"
git push -u origin feature/login-form   # set upstream tracking branch
```

Keep a feature branch current with `main` via rebase (clean, linear history):

```
git switch feature/login-form
git fetch origin
git rebase origin/main                  # replay my commits on top of latest main
# resolve any conflicts, then:
git push --force-with-lease             # safe force: fails if remote moved unexpectedly
```

Resolve a merge conflict:

```
git switch feature/login-form
git merge main
# Git marks conflicts in files with <<<<<<< ======= >>>>>>>
#   edit each conflicted file to the correct combined result
git add lib/features/auth/login_form.dart   # mark as resolved
git commit                                   # completes the merge
git merge --abort                            # or bail out entirely
```

Undo safely on a shared branch vs. locally:

```
git revert a1b2c3d           # new commit undoing a1b2c3d (safe on main)
git reset --hard HEAD~1      # drop last commit locally (never on shared history)
git reflog                   # find and recover a lost commit
git reset --hard HEAD@{2}
```

A clean Pull Request flow:

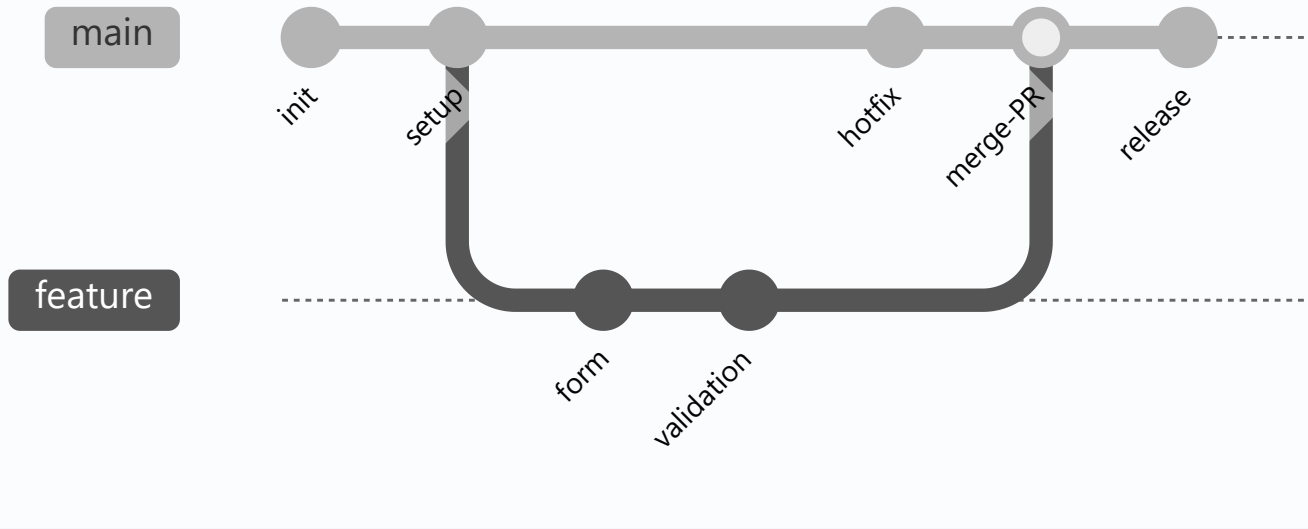
```
git switch main && git pull --ff-only
git switch -c fix/cart-total-rounding
# make one focused change + tests
git add -A && git commit -m "Fix cart total rounding to 2 decimals"
git push -u origin fix/cart-total-rounding
# open PR, request review, address feedback with fixup commits:
git commit --fixup HEAD
git rebase -i --autosquash origin/main   # tidy before merge
git push --force-with-lease
```

Tagging a release:

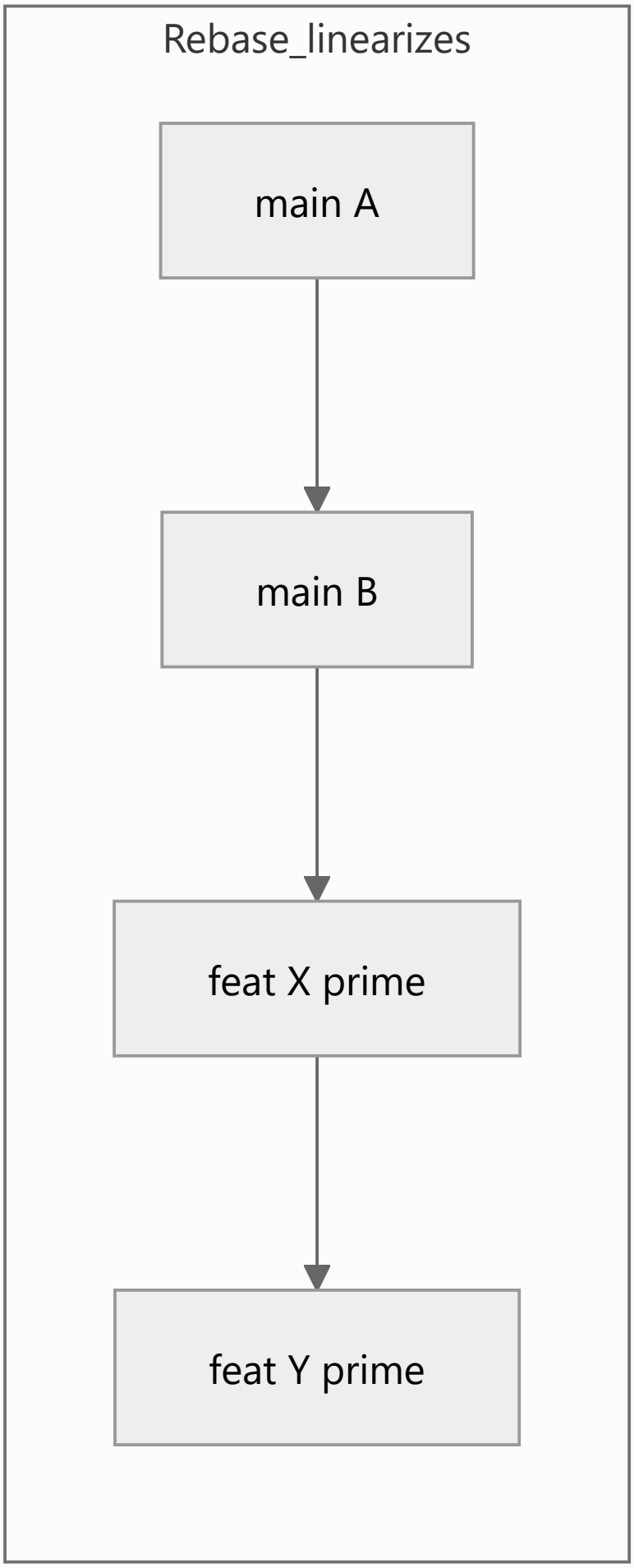
```
git tag -a v1.4.0 -m "Release 1.4.0: checkout revamp"
git push origin v1.4.0
```

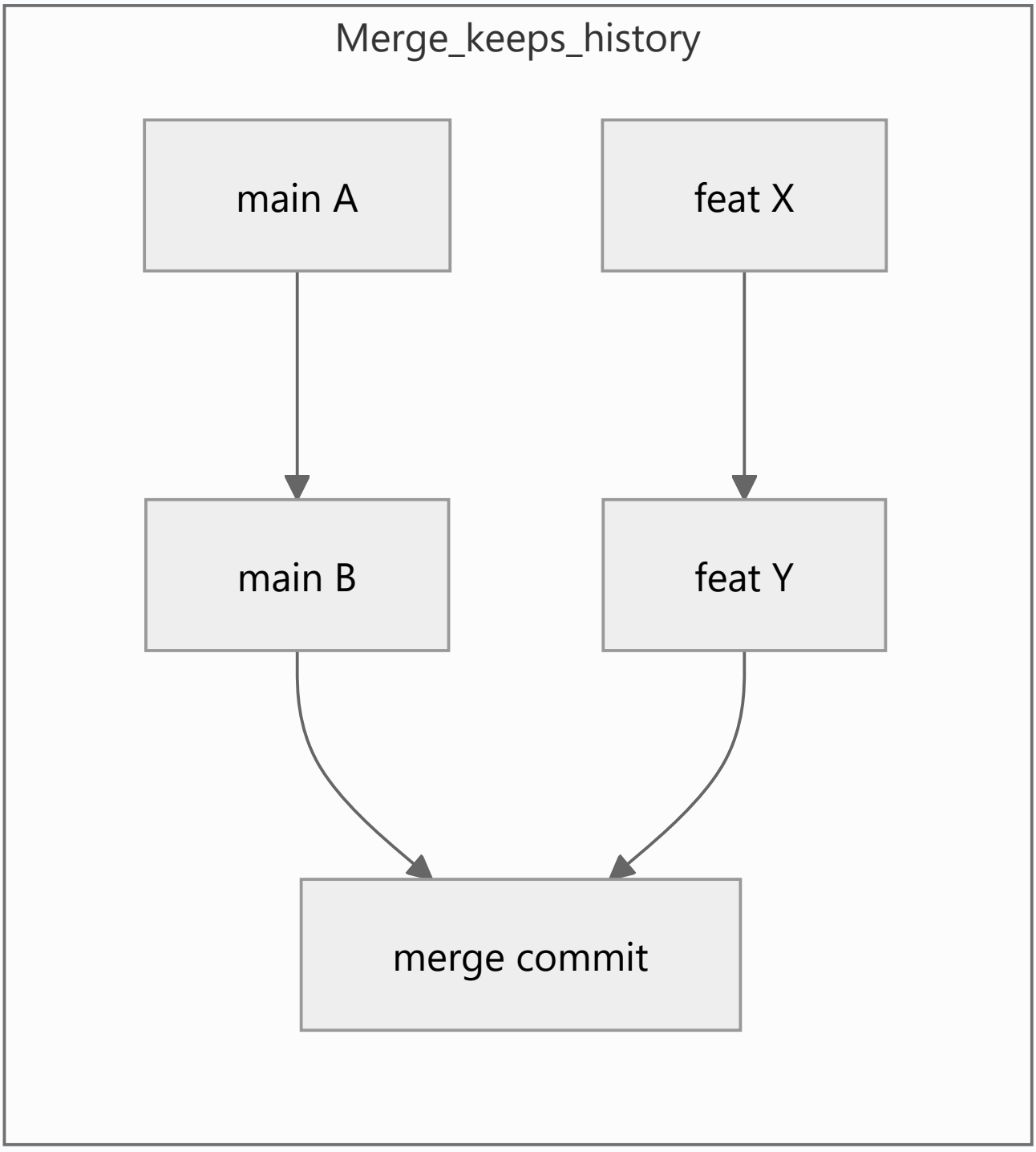
Diagrams

Merge vs rebase, and a trunk-based-style branching flow:



Conceptually, **merge** keeps both histories and adds a join point, while **rebase** rewrites the feature commits onto the new base to produce a straight line:


Rebase_linearizes
main A
main B
feat X prime
feat Y prime



Common Mistakes

- **Committing secrets** (API keys, `.env`, keystores). Correction: add them to `.gitignore` *before* the first commit; if already committed, rotate the secret immediately and purge with `git filter-repo` — deleting the file in a new commit does NOT remove it from history.
- **Force-pushing to a shared branch** (`git push -f` to `main`). Correction: never force-push shared branches; on your own branch use `--force-with-lease`, which refuses to clobber

commits you have not seen.

- **Giant PRs** (5000 lines, 40 files, mixed concerns). Correction: keep PRs small and single-purpose; split refactors from features. Reviewers approve small PRs carefully and rubber-stamp huge ones.
- `git reset --hard` **losing work.** Correction: check `git stash` or `git reflog`; prefer `git revert` for shared history.
- **Committing generated/build artifacts** (`build/`, `.dart_tool/`, `*.g.dart` if generated in CI). Correction: gitignore them; commit only source.
- **Vague messages** ("fix", "wip", "update"). Correction: write an imperative subject explaining the change and a body explaining *why*.
- **Merging without pulling** and pushing stale refs. Correction: `git fetch` and rebase/merge before pushing.
- **One branch reused for many features.** Correction: one branch per unit of work.

Best Practices

- Commit **atomically**: one logical change per commit, with tests included in the same commit.
- Write **imperative** subject lines ("Add", "Fix", "Remove") under ~50 chars, blank line, then a body explaining rationale and tradeoffs.
- Keep branches short-lived; integrate with mainline daily to avoid divergence pain.
- Protect `main` with branch protection: require PR review, green CI (see CI pipeline and automation), and up-to-date branches.
- Use `--force-with-lease`, never `--force`, and never rewrite public history.
- Maintain a curated `.gitignore` (start from a language/framework template) and a `.gitattributes` for line endings and LFS.
- Tag releases with **semantic versioning** and let CI cut releases from tags (see delivery and release).
- Review PRs against the team's checklist (see code quality and review).

Performance

- **Repo size** grows with history and large binaries. Blobs are deduplicated, but a big file committed once stays in history forever unless rewritten.
- **Packfiles + delta compression** (`git gc`, `git repack`) shrink the object store dramatically and speed up transfers; run gc periodically (Git auto-gcs).
- **Shallow clone** (`git clone --depth 1`) fetches only recent history — huge win for CI where full history is unneeded.

- **Partial/sparse checkout** (`--filter=blob:none` , `git sparse-checkout`) helps with large monorepos, fetching blobs/paths on demand.
- **Large files**: binaries bloat every clone. Use **Git LFS**, which stores a small pointer in Git and the blob in a separate store, keeping the DAG lean.
- Fewer, well-packed objects and shorter histories in CI reduce clone time from minutes to seconds.

Advantages

- Distributed: every clone is a full backup; most operations are local and fast.
- Immutable, tamper-evident history via content addressing.
- Cheap branching/merging enables true parallel development.
- Rich ecosystem (hosting, CI, code review, hooks).
- Powerful history tools: `bisect` , `blame` , `reflog` , `cherry-pick` .

Disadvantages

- Steep mental model; commands are inconsistent and easy to misuse destructively.
- Poor at large binaries without LFS.
- Rewriting history is dangerous and confuses collaborators.
- Very large monorepos strain vanilla Git without extra tooling (scaling, VFS, sparse checkout).

Interview Questions

🟢 **1. Is a Git commit a diff or a snapshot?** A snapshot. A commit points to a full root tree describing every file's state. Diffs are computed on demand by comparing two trees. Packfiles use delta compression internally, but that is a storage optimization, not the model.

🟢 **2. What is the difference between `git fetch` and `git pull` ?** `fetch` downloads remote objects and updates remote-tracking refs (`origin/main`) without touching your working branch. `pull` is `fetch` followed by a `merge` (or `rebase` with `--rebase`) into your current branch.

🟢 **3. What is the staging area (index)?** An intermediate layer between the working tree and the repository that holds the exact tree you will commit next. It lets you craft commits selectively with `git add` rather than committing everything changed.

🟡 **4. `git merge` vs `git rebase` — when do you use each?** Merge preserves true history and creates a merge commit; use it to integrate completed branches, especially shared ones. Rebase rewrites your commits onto a new base for linear history; use it to keep a *local, unpushed* feature branch current. Never rebase commits others have based work on.

🟡 **5. `git reset` vs `git revert`?** `reset` moves a branch pointer (optionally altering index/worktree) and rewrites history — appropriate locally. `revert` creates a new commit that undoes a prior commit, preserving history — the safe choice on shared branches.

🟡 **6. What does `--force-with-lease` do that `--force` doesn't?** It refuses the push if the remote branch has advanced beyond what your local remote-tracking ref recorded, preventing you from silently overwriting a teammate's pushed commits. Plain `--force` clobbers unconditionally.

🟡 **7. How does Git know two files with the same content are identical?** Content addressing: the file's bytes hash to the same SHA, producing one blob object. This gives automatic deduplication across the repo and history.

🔴 **8. A secret was committed and pushed. What now?** Treat it as compromised: rotate/revoke the credential immediately. Then purge it from history (`git filter-repo` or BFG), force-push, and have everyone re-clone. Deleting the file in a new commit is insufficient — it remains in every ancestor.

🔴 **9. Explain fast-forward vs. no-fast-forward merges.** If the target branch tip is an ancestor of the source, Git can just move the pointer forward (fast-forward), producing no merge commit and a linear history. `--no-ff` forces a merge commit even when a fast-forward is possible, preserving an explicit record that a branch was merged.

🔴 **10. How would you find which commit introduced a bug across 500 commits?** `git bisect`: mark a known-good and known-bad commit; Git binary-searches, checking out midpoints for you to test. In ~log2(N) steps it pinpoints the offending commit. Can be automated with `git bisect run <test-script>`.

🔴 **11. What keeps an object from being garbage-collected?** Reachability from a ref (branch, tag, HEAD, stash, or reflog entry). Unreachable objects past the grace period are removed by `git gc`. This is why the reflog can rescue "lost" commits.

🟡 **12. Monorepo vs polyrepo — one tradeoff each way.** Monorepo: atomic cross-project changes and unified tooling, but needs scaling tooling and disciplined CI. Polyrepo: strong isolation and independent release cadence, but cross-cutting changes span many repos and dependency/version coordination gets hard.

Senior Engineer Tips

- Learn the object model once; every command becomes predictable. `git cat-file -p HEAD` and `git log --graph --oneline --all` are your microscopes.
- Configure `pull.rebase = true` (or `--ff-only`) to avoid accidental merge-commit noise on pulls.
- Use `git commit --fixup` + `rebase -i --autosquash` to keep review history clean without manual squashing.
- Set up `git rerere` so repeated conflict resolutions are remembered and reapplied.

- Use `git worktree` to check out multiple branches simultaneously without stashing.
- Keep `.gitattributes` authoritative for line endings (`* text=auto`) to end CRLF/LF wars across OSes.
- Sign commits/tags (GPG/SSH) where provenance matters.

Architect Perspective

At org scale the branching model and repo topology are strategic:

- **Trunk-based development** with short-lived branches and feature flags scales best for continuous delivery: everyone integrates to `main` many times a day, CI enforces quality, and releases are cut from trunk. It minimizes merge debt and pairs naturally with CI/CD.
- **Git Flow** suits products with scheduled, versioned releases and long support windows (e.g. desktop/on-prem), but its many long-lived branches add integration overhead and slow feedback.
- **GitHub Flow** is the pragmatic middle: `main` is always deployable, work happens on branches merged via PR.

Monorepo vs polyrepo is an org-design decision. A monorepo (managed with tooling like Melos for Dart — see monorepo and Melos) gives atomic cross-package refactors, one source of truth, and shared CI, at the cost of needing sparse checkout, affected-package build graphs, and strong CI to keep the trunk green. Polyrepos give teams autonomy and blast-radius isolation but push integration complexity into dependency/version management. Choose based on how coupled your components are and how much platform tooling you can invest in.

Concern	Git Flow	GitHub Flow	Trunk-Based
Long-lived branches	Many (develop, release, hotfix)	One (main) + short PR branches	Only main; branches live hours/days
Release model	Scheduled, versioned	Continuous	Continuous, feature-flag gated
Merge/integration cost	High	Low	Lowest
Best for	Versioned/on-prem products	Web apps, most teams	High-velocity CD orgs
CI reliance	Moderate	High	Very high

Summary

Git is a content-addressable object store (blobs, trees, commits, tags) shaped into an immutable commit DAG, with branches, tags, and HEAD as cheap movable pointers and the index as the staging layer between working tree and repository. Understanding this model demystifies every operation: commits add nodes, branches move pointers, merges join histories, rebases replay commits as new nodes, and resets/reverts differ only in whether they rewrite history.

Professionally, teams pair this model with a branching strategy (usually GitHub Flow or trunk-based), small atomic commits, disciplined PRs, protected mainlines, and CI to keep a shared codebase moving fast and safely.

Revision Notes

- Four objects: **blob** (bytes), **tree** (dir listing), **commit** (snapshot + parents), **tag**.
- Everything is addressed by SHA of its content → dedup + tamper-evidence.
- Three stages: **working tree** → **index (staging)** → **repository**.
- Branch = pointer to a commit; HEAD = pointer to current branch; refs are files holding a hash.
- Merge = new commit, keeps history; rebase = new commits, linear, rewrites history.
- reset moves the branch (rewrites); revert adds a commit (safe on shared).
- `--force-with-lease` over `--force`; never rewrite public history.
- Objects: loose → packed via `gc`; deltas are a storage optimization.
- SemVer: MAJOR.MINOR.PATCH; tag releases.

Practice Questions

1. Draw the object graph produced by two commits where only one file changed. How many blob objects exist?
2. Explain, in pointer terms, exactly what `git switch -c feature` and then `git commit` do to refs and HEAD.
3. When is a merge a fast-forward, and how do you force a merge commit anyway?
4. Why does deleting a committed secret in a new commit fail to secure it? What is the correct remediation?
5. Contrast the disk representation of a fresh repo (loose objects) with one after `git gc`.
6. Give a scenario where rebasing is the wrong choice and merging is correct.

Coding Questions

1. **Inspect the store.** In any repo, run `git cat-file -p HEAD`, then follow the tree hash with `git cat-file -p <tree>`, then a blob. *Acceptance:* you can name the type and content of each object and explain how a commit reaches a file's bytes.
2. **Craft two atomic commits.** Make two unrelated one-line changes and commit them separately using `git add -p`. *Acceptance:* `git log --oneline` shows two focused commits with imperative messages; neither mixes concerns.
3. **Rebase cleanly.** Create a feature branch, add two commits, add a commit to `main`, then rebase the feature onto `main`. *Acceptance:* `git log --graph --oneline --all` shows linear history with new SHAs for the feature commits.

4. **Resolve a real conflict.** Create divergent edits to the same line on two branches and merge them. *Acceptance:* conflict markers resolved, `git status` clean, resulting file correct.
5. **Recover a lost commit.** `git reset --hard HEAD~1`, then restore it via `git reflog`. *Acceptance:* the "lost" commit is back on a branch.
6. **Squash before merge.** Use `git commit --fixup` and `git rebase -i --autosquash`. *Acceptance:* review branch collapses to intended commits.

Mini Project

Team Git simulation for a Flutter feature.

Build a small Flutter package repo and simulate a two-developer workflow end to end:

1. Initialize the repo, add a sensible Flutter `.gitignore` (ignore `build/`, `.dart_tool/`, `*.iml`, local env files), and make an initial commit tagged `v0.1.0`.
2. Enable trunk-based discipline: protect `main` conceptually (only merge via PR), and create two feature branches representing two developers editing overlapping files (e.g. both touch `lib/main.dart` and a shared widget).
3. Developer A merges first via a fast-forward-avoided PR (`--no-ff`). Developer B rebases onto the updated `main`, resolves the resulting conflict, and opens a clean PR with an atomic history (use `--autosquash`).
4. Introduce a deliberate regression, then locate it with `git bisect run` against a test script.
5. Cut a release: bump to `v0.2.0` following SemVer, tag it, and write CHANGELOG entries. Wire a placeholder CI check (see CI pipeline and automation) that must pass before merge.

Acceptance criteria: `git log --graph --oneline --all` shows a coherent, mostly-linear history; no build artifacts or secrets are tracked; both PRs are small and single-purpose; the regression commit is identified by bisect; `v0.1.0` and `v0.2.0` tags exist and follow SemVer; the review process matches code quality and review and release steps match delivery and release.

Agile & Team Collaboration

> Agile is a set of values and iterative practices that lets a team deliver working software in short feedback loops, adapting to change instead of committing to a fixed plan made when the team knew the least.

Introduction

Agile is not a tool, a certification, or a Jira board. It is a way of organizing engineering work around **short cycles, frequent delivery, and continuous feedback**. The bet is simple: for software, the plan is always wrong at the start, so you should optimize for *changing the plan cheaply* rather than for *executing a plan perfectly*.

This handbook entry is a **practice** topic, not a language topic. There is no bytecode, no widget tree, no garbage collector. Instead we treat the *process itself* as the system: the "state" is where work lives (backlog, board columns), the "runtime" is what happens during a sprint or a day, and "performance" is team throughput. We cover Scrum, Kanban, estimation, user stories, Definition of Done, collaboration mechanics, and code ownership — then trace a single Flutter feature from ticket to production.

Related practice topics:

- Leadership & Mentorship
- Version Control with Git
- Code Quality & Review
- Testing Fundamentals & the Pyramid

Why this concept exists

Waterfall failed for software. The waterfall model — requirements, then design, then implementation, then testing, then release, each phase completed before the next begins — was borrowed from construction and manufacturing. It assumes the requirements are knowable up front and stable. For software, both assumptions are false:

- **Requirements are discovered, not specified.** Users cannot tell you what they want until they see something. The most expensive requirements bugs are the ones baked into a spec that nobody validated for six months.
- **The cost of change grows exponentially in waterfall.** A requirement change after the design is frozen forces rework of everything downstream. So waterfall projects resist change — and ship the wrong thing, on time.

- **Integration is deferred to the end**, where all the risk piles up. The "90% done for 90% of the time" problem.
- **Feedback arrives too late to act on.** You learn the product is wrong at the demo, after the budget is spent.

Agile exists to invert this. Instead of one long bet, you make many small bets. You build a thin working slice, show it to real users, and let what you learn reshape the next slice. **Change stops being a failure of planning and becomes the expected input to planning.**

Why story points beat hours (previewing the estimation section): hours are an absolute promise about the future that a person is bad at making and gets punished for missing. Points are a *relative* measure of size/complexity/risk that the team calibrates against itself. Points let you forecast with *velocity* (empirical throughput) instead of heroics, and they de-couple "how big" from "how long," which is exactly the coupling that makes hour-estimates a lie.

Real-world analogy

Cooking a tasting menu for a guest whose tastes you don't know.

Waterfall: interview the guest once, then disappear into the kitchen for eight hours and serve all twelve courses at once. If they hate the salt level, you find out after everything is plated and cold.

Agile: serve one small course, watch their face, adjust the seasoning, serve the next. Each course is a **sprint**. The reactions are **feedback**. The running menu you keep re-prioritizing is the **backlog**. You never cook more than you can serve and taste in one sitting — that limit is your **WIP limit**. By the end, the meal fits *this* guest, because you steered continuously instead of guessing once.

Problem Statement

A team of five is building a Flutter app. The stakeholders keep changing their minds. Some features turn out harder than expected; others turn out unnecessary once seen on-screen. Management wants a date. Engineers want to not be blamed for a date invented before the code existed. QA keeps receiving half-finished features called "done." Two developers keep editing the same files and stepping on each other.

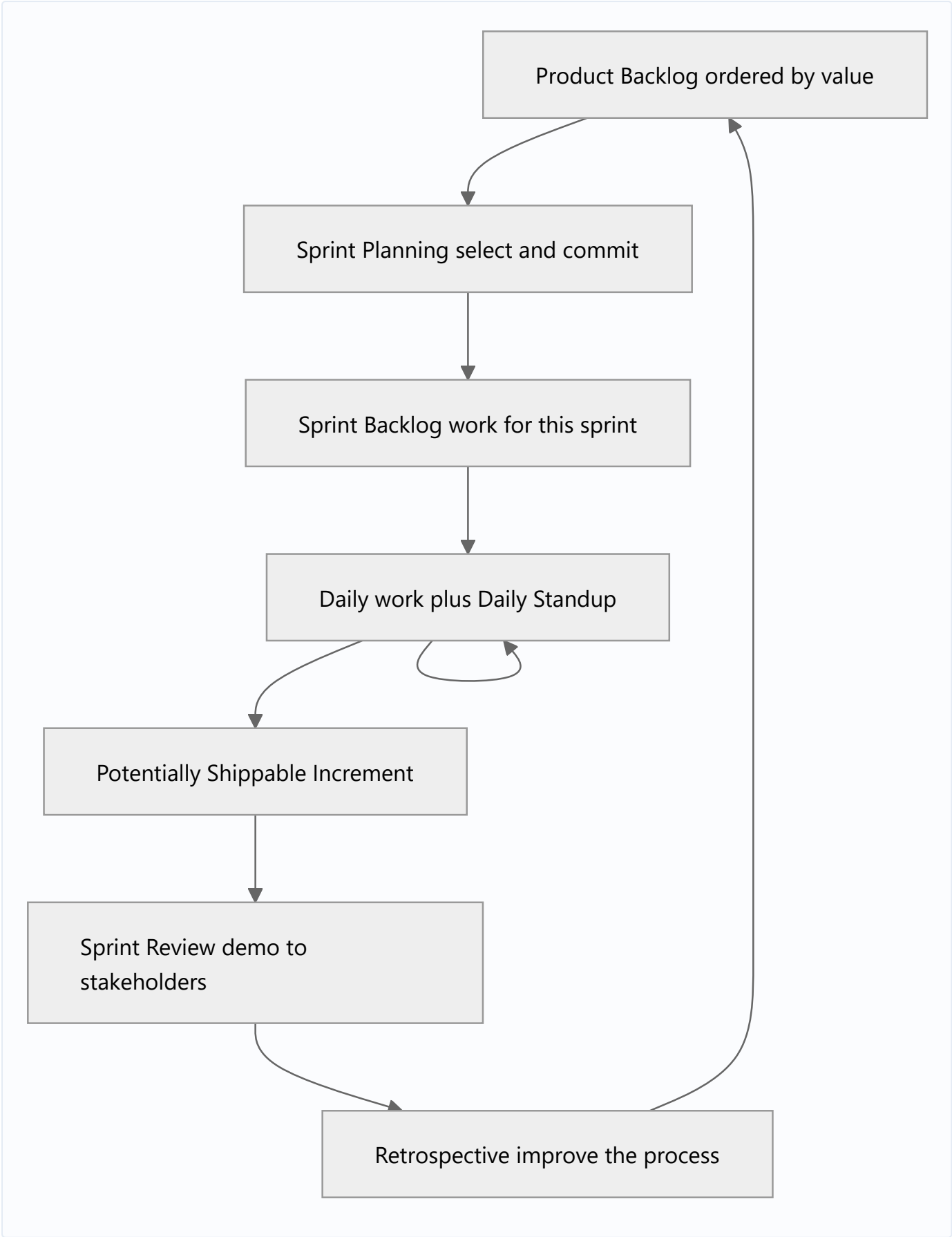
We need a process that:

1. Delivers something usable and demonstrable at a regular cadence.
2. Makes forecasting honest and data-driven rather than a guess under pressure.
3. Defines unambiguously when work is *actually* finished.
4. Coordinates who works on what and who owns which code.
5. Absorbs changing requirements without collapsing.

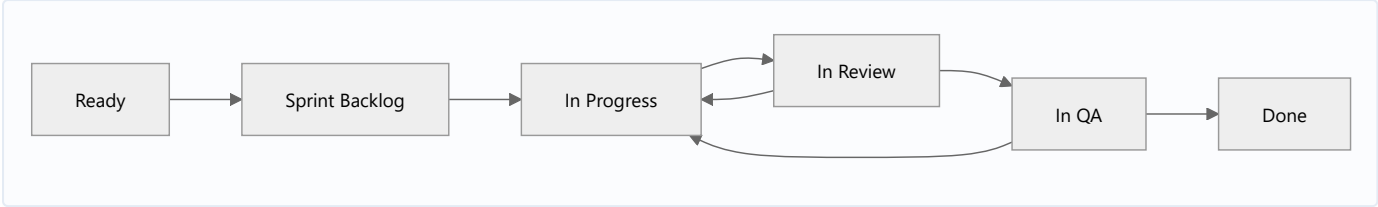
Agile — usually via Scrum or Kanban — is the answer, and this document is the map.

Internal Working

The core of Scrum is a repeating loop. Work flows from an ordered **product backlog**, a slice is pulled into a **sprint**, built, reviewed, and shipped as an **increment**, then the team reflects and repeats.



A single story also moves through **states** on the board:



Each state transition is a handoff with an entry condition. A story cannot enter **In Progress** unless it met the **Definition of Ready**; it cannot reach **Done** unless it met the **Definition of Done**.

Memory Representation

Repurposed: *where the state of work physically lives.*

"Memory location"	What it holds	Volatility
Product backlog	Every known desirable piece of work, ordered by value. The long-term store.	Constantly re-prioritized; never "full"
Sprint backlog	The subset the team committed to *this* sprint, plus the plan to build it.	Fixed at planning; owned by the dev team
Board columns (To Do / In Progress / Review / Done)	The live position of each work item — like a program counter per story.	Changes many times a day
WIP limit	The max cards allowed in a column at once.	A constraint, not storage — it caps concurrency
Increment	The sum of all Done items, integrated and shippable.	Append-only; grows each sprint

The key insight: the board is the team's **shared working memory**. A card in "In Progress" with no owner and no movement for three days is a *leaked reference* — work that is allocated but not running.

Compiler Behavior

Not applicable — because Agile is an organizational process, not source code. Nothing here is compiled, type-checked, or lowered to an intermediate representation. The closest analogue is a linting/validation step: a Definition of Ready acts like a compile-time gate that rejects malformed work items before they enter the sprint, and a CI pipeline acts like the actual compiler for the *code* the process produces (see Version Control with Git).

Runtime Behavior

Repurposed: *what actually happens during a sprint and during a day.*

A day (the daily stand-up, ~15 min, same time, standing up): each dev answers three questions in service of one goal — *are we still on track to meet the sprint goal?* Not "what did you do" for a manager, but "what is blocking us and where do we need to swarm." Blockers surface here and get an owner; deep discussions are taken *offline* ("let's park that and take it after").

A sprint (typically 2 weeks):

- Day 1: **Sprint planning.** PO presents top backlog items; team clarifies, estimates, and commits to a **sprint goal** and a sprint backlog it believes is achievable.
- Days 1–9: build. Continuous integration, code review, testing. The burndown chart tracks remaining work.
- Mid-sprint: **backlog refinement** (grooming) — the PO and team prepare and estimate upcoming stories so the *next* planning is fast.
- Day 10: **Sprint review** (demo the increment to stakeholders; collect feedback into the backlog) followed by **retrospective** (the team inspects its own process and picks 1–2 concrete improvements).

The sprint is **time-boxed**: the date is fixed, the scope flexes. If work won't fit, you cut scope, not quality, and never silently extend.

Flutter Engine Behavior (if applicable)

Not applicable — because the Flutter engine renders widgets and drives the raster/UI threads; it has no notion of sprints, backlogs, or ceremonies. Process practices influence *what* the engine eventually runs and *how confidently*, but they do not execute on it.

Dart VM Behavior (if applicable)

Not applicable — because the Dart VM executes Dart code (JIT in debug, AOT in release). Agile governs the humans who write that code, not the VM. No isolate, no GC pause, no JIT tier corresponds to a stand-up or a retro.

Examples

A well-formed user story with acceptance criteria

```
Title: Offline caching of the product catalog

As a shopper on a flaky connection,
I want the product catalog to load from a local cache when I'm offline,
so that I can keep browsing without a network round-trip.

Acceptance Criteria (Gherkin style):
  Given I have opened the catalog at least once while online
    And I am now offline
   When I open the catalog screen
   Then I see the last-synced products within 300 ms
    And a banner reads "Offline - showing cached data"

  Given I am offline
   When I pull to refresh
   Then the refresh fails gracefully with a retry affordance
    And no data is lost

Out of scope: offline mutations (add-to-cart while offline) - tracked separately.
```

This story is **INVEST**: Independent, Negotiable, Valuable, Estimable, Small, Testable.

A Definition of Done checklist

```
Definition of Done (applies to EVERY story before it can be "Done")
[ ] Code merged to main via reviewed PR (>=1 approval)
[ ] Meets acceptance criteria (verified by PO or QA)
[ ] Unit + widget tests written; coverage not decreased
[ ] Analyzer clean (flutter analyze, no new warnings)
[ ] Formatted (dart format) and CI green
[ ] No new TODOs without a linked ticket
[ ] Documentation / changelog updated if user-facing
[ ] Feature behind a flag if partially rolled out
[ ] Deployed to staging and smoke-tested
```

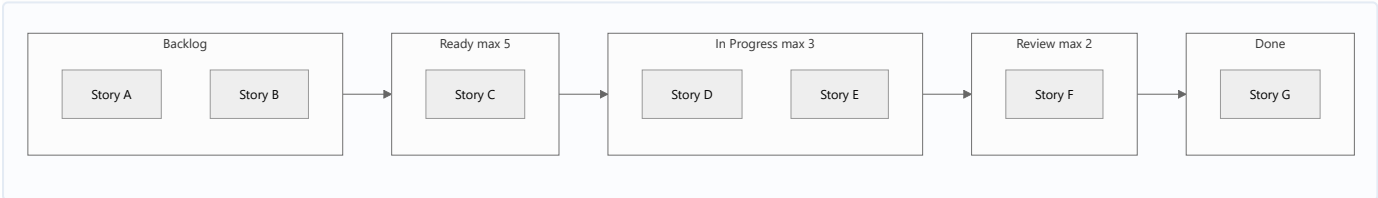
A CODEOWNERS snippet (`.github/CODEOWNERS`)

```
# Default owners for everything
*                       @org/mobile-core

# Feature areas
/lib/features/catalog/  @org/catalog-team
/lib/features/payments/ @org/payments-team @jane-lead

# Cross-cutting concerns need extra eyes
/lib/theme/             @org/design-system
/pubspec.yaml           @org/mobile-core @org/release-eng
**/*.arb                @org/localization
```

A matching PR touching `/lib/features/payments/` will auto-request review from `@org/payments-team` and `@jane-lead`, enforcing **weak ownership** with a required approval gate.

Diagrams

Kanban board with WIP limits:



When "In Progress" hits its WIP limit of 3, nobody pulls a new card — instead the team **swarms** the card closest to Done. Limiting WIP is how Kanban converts "everyone is busy" into "work actually finishes."

Common Mistakes

Mistake	Why it hurts	The fix
Stand-up as status theater — reporting to a manager, one by one, robotically	Wastes 15 min/day, surfaces no blockers, treats people as progress bars	Direct it at the *sprint goal* and blockers; take details offline; skip if nothing to coordinate
Estimating in hours	Punishes people for being wrong about the future; couples size to speed	Estimate relative size in **points**; forecast with velocity
No Definition of Done	"Done" means five different things; bugs and tech debt ship silently	Write an explicit, checkable DoD the whole team agrees to
Sprints that always overflow	Team over-commits; morale erodes; forecasts become fiction	Commit to less; use *actual* velocity, not aspirational
Backlog is a junk drawer	800 stale items, nothing prioritized, planning is chaos	PO ruthlessly orders and prunes; refine continuously
Retro with no action	Same problems every sprint; team stops believing in retros	Pick 1–2 concrete actions with owners; review them next retro
Changing scope mid-sprint	Destroys the one stable box the team has	Protect the sprint; new work goes to the backlog for next planning
Story "won't fit" so it lives across sprints	Zero delivered value, no feedback, invisible progress	Split it vertically into shippable slices

Best Practices

- **Slice stories vertically**, not by layer. A slice should go from UI to data and deliver observable value, not "build the repository layer" (invisible to users).
- **Keep the sprint goal singular and memorable.** If you can't state it in one sentence, you committed to too much.
- **Refine continuously** so planning is a confirmation, not a discovery session.
- **Make the board honest.** A card's column must reflect reality *now*. Stale boards rot trust.
- **Separate Definition of Ready from Definition of Done** — one gates entry, one gates exit.
- **Automate the DoD where possible** — CI enforces tests, format, analyzer, coverage so "done" isn't a matter of opinion.
- **Timebox ceremonies** and start on time regardless of who is late.
- **Prefer async by default, sync for ambiguity** (see Collaboration mechanics below).

Collaboration mechanics

- **Async vs sync:** Default to async (written PRs, docs, threaded discussion) — it scales across time zones, creates a record, and protects deep-work focus. Switch to sync (call, pairing) when the topic is ambiguous, emotionally charged, or ping-ponging past ~3 round trips.

- **RACI** clarifies decisions: **R**esponsible (does the work), **A**ccountable (one person, owns the outcome), **C**onsulted (two-way input), **I**nformed (one-way notice). Exactly one A per decision.
- **Pair programming:** two devs, one keyboard — great for hard problems, onboarding, and knowledge transfer; raises the **bus factor**.
- **Mob programming:** the whole team on one problem — reserved for the highest-risk or highest-alignment work.

Performance

Repurposed: *team flow metrics*. You optimize the *system of work*, not individual utilization.

Metric	Definition	What it tells you
Velocity	Points completed per sprint (rolling average)	Forecasting capacity — *not* a productivity KPI to compare across teams
Cycle time	Time from "In Progress" to "Done"	How fast work flows once started; the number to shrink
Lead time	Time from request created to delivered	The customer's experience of speed
Throughput	Items completed per unit time	Delivery rate, complements velocity
Flow efficiency	Active time / total time (active + waiting)	Often <15% — most time is *waiting*, not working. Attack the waits (review queues, handoffs)
WIP	Items in progress right now	High WIP = high cycle time (Little's Law: `Cycle time = WIP / Throughput`)

The counter-intuitive lesson: to go faster, **lower WIP**. Finish before you start. Utilization near 100% *increases* delay, exactly like a full disk or a saturated CPU queue.

Advantages

- Delivers usable software early and often; risk is discovered while it's cheap.
- Absorbs changing requirements as normal input.
- Tight feedback loops steer toward the *right* product, not just a shipped one.
- Empirical forecasting (velocity) beats up-front guessing.
- Improves morale and ownership — the team commits to its own plan and improves its own process.
- Continuous integration keeps the product shippable, avoiding big-bang integration risk.

Disadvantages

- **Cargo-culting**: teams adopt the rituals without the values and get meetings without benefit.
- Requires genuine stakeholder availability; an absent PO starves the backlog.

- Weak fit for truly fixed-scope, fixed-date, fixed-contract work (some regulated/hardware contexts).
- Metrics are easily weaponized (velocity as a stick), which corrupts the estimates.
- Ceremony overhead can dominate for tiny teams; discipline is required to keep it lightweight.
- Long-term architecture can be neglected if every sprint chases the nearest feature.

Interview Questions

🟢 **1. What are the four values of the Agile Manifesto?** Individuals and interactions **over** processes and tools; working software **over** comprehensive documentation; customer collaboration **over** contract negotiation; responding to change **over** following a plan. Crucially: the items on the right have value, but we value the items on the left *more*.

🟢 **2. Scrum vs Kanban — when would you pick each?** Scrum: time-boxed sprints, fixed roles/ceremonies, commitment to a sprint goal — good when work is plannable in batches and stakeholders want a cadence. Kanban: continuous flow, WIP limits, no sprints, pull-based — good for unpredictable, interrupt-driven work like support or ops. Many teams run "Scrumban."

🟢 **3. What is a story point and why not estimate in hours?** A point is a relative measure of size/complexity/uncertainty, calibrated to the team. Hours are absolute promises humans estimate poorly and get punished for missing; points decouple size from speed and let velocity do the forecasting empirically.

🟡 **4. Explain Definition of Done vs Definition of Ready.** DoR gates *entry*: a story is ready to be worked (clear, estimated, dependencies known, acceptance criteria written). DoD gates *exit*: objective, checkable conditions every story must meet to be called done (merged, tested, CI green, deployed to staging, criteria verified). One prevents starting garbage; the other prevents shipping garbage.

🟡 **5. What makes a good user story? (INVEST)** Independent, Negotiable, Valuable, Estimable, Small, Testable. Written as "As a [role] I want [capability] so that [benefit]," with explicit acceptance criteria. It should deliver a vertical slice of value, not a horizontal technical layer.

🟡 **6. What are WIP limits and why do they speed a team up?** A cap on concurrent work per board column. By Little's Law, cycle time = WIP / throughput, so cutting WIP cuts cycle time. Lower WIP forces finishing over starting, exposes bottlenecks, and reduces context-switching — the team delivers faster by doing fewer things at once.

🔴 **7. Behavioral: How do you handle a story that won't fit in a sprint?** First, don't carry it across sprints — that delivers zero feedback. Split it **vertically** into thin end-to-end slices that each ship value (e.g., "show cached list read-only" before "cache with refresh" before "offline

mutations"). If it genuinely can't be split, it's a spike/epic: create a time-boxed research spike to reduce uncertainty, then decompose. Escalate to the PO to re-prioritize the slices. The anti-pattern is a giant WIP card dragging across three sprints.

🔴 **8. Behavioral: A stakeholder wants to add scope mid-sprint. What do you do?** Protect the sprint. Acknowledge the value, put it in the backlog, and make the trade-off visible: adding it now means dropping something of equal size (the PO decides which). If it's a true emergency (production incident), that's a different track — abort/replan the sprint transparently rather than quietly overloading the team. The sprint's stability is what makes forecasting possible.

🟡 **9. Explain code ownership models.** Strong: only the owner edits their files. Weak: an owner exists but others may contribute with review. Collective: anyone may change anything, guarded by tests, standards, and review. Collective maximizes bus factor and flow but needs strong CI discipline; strong ownership creates deep expertise but bottlenecks and single points of failure.

🟡 **10. What is bus factor and how do you raise it?** The number of people who can be "hit by a bus" before the project stalls — i.e., how concentrated critical knowledge is. Raise it with pairing/mobbing, rotating ownership, documentation, collective ownership, and CODEOWNERS that list teams rather than individuals.

🔴 **11. Your velocity is dropping. How do you investigate?** Don't treat velocity as productivity — investigate flow. Check cycle time and flow efficiency: is work stuck in review/QA (a bottleneck, not slow devs)? Is WIP too high? Did estimates inflate under pressure? Was the team interrupted by unplanned work or attrition? Look at the retro history. Velocity is a planning tool; a drop is a signal to find waits, not to push people.

🔴 **12. Behavioral: The daily stand-up feels useless. How do you fix it?** Diagnose the failure mode. If it's status theater, re-anchor it on the sprint goal and blockers, not on reporting to a manager; take detailed discussions offline; walk the board right-to-left (focus on finishing). If truly nothing needs coordinating on a given day, skip it — the ritual serves the team, not vice versa.

Senior Engineer Tips

- **Estimate uncertainty, not effort.** A point spike usually means "we don't understand this yet" — resolve the unknown with a spike before committing.
- **The board is a diagnostic instrument.** Read it right-to-left: cards piling up in Review mean your bottleneck is review capacity, not coding — swarm there.
- **Say "no" via the backlog, not in the moment.** "Great idea — let's rank it against everything else" reframes scope creep as prioritization.
- **Automate the boring half of the DoD.** Anything a human "remembers to check" will eventually be forgotten; move it into CI. See Code Quality & Review.
- **Protect deep work.** Default async; batch your interruptions; a fragmented senior is a bottleneck multiplier.

- **Vertical slices always.** If a story has no demoable outcome, it's a task, not a story.
- **Retro actions need owners and a due date**, or they're wishes.

Architect Perspective

At one team, Scrum "just works." Across dozens of teams, the hard problems are **dependencies and communication topology**, not ceremonies.

- **Conway's Law:** "Organizations design systems that mirror their communication structure." If three teams own one screen, you'll get three seams in that screen. So architects use the **Inverse Conway Maneuver**: shape team boundaries to match the *desired* architecture. Team topology *is* system design.
- **The Spotify model** (Squads, Tribes, Chapters, Guilds) is one attempt at scaling autonomy: squads are autonomous cross-functional teams; chapters/guilds share expertise horizontally so autonomy doesn't fragment standards. Treat it as inspiration, not a template — Spotify itself doesn't run it verbatim.
- **Cross-team dependencies are the enemy of flow.** Every hand-off adds queue time (flow efficiency drops). Architect for **loosely coupled, independently deployable** modules so teams don't block each other — the same modularity argument as clean architecture, applied to org design.
- **Alignment vs autonomy:** high autonomy without alignment yields divergence; high alignment without autonomy yields bottlenecks. The architect's job is to set clear standards (alignment) and then get out of the way (autonomy) — mirroring the mentorship stance in Leadership & Mentorship.
- **Scaling frameworks** (SAFe, LeSS, Scrum@Scale) formalize cross-team planning, but each adds process weight; adopt only the coordination you actually need.

Summary

Agile replaced waterfall because software requirements are discovered, not specified, and change is cheaper to embrace than to resist. **Scrum** organizes work into time-boxed sprints with clear roles (PO owns *what*, Scrum Master owns *how the process runs*, dev team owns *how it's built*), artifacts (product backlog, sprint backlog, increment), and ceremonies (planning, stand-up, review, retro). **Kanban** optimizes continuous flow with WIP limits. Estimation uses relative **story points** and empirical **velocity** rather than dishonest hour promises. **User stories** capture value ("As a… I want… so that…") and are gated by **Definition of Ready** on entry and **Definition of Done** on exit. Collaboration balances async and sync, uses RACI for decisions, and raises **bus factor** through pairing and collective ownership backed by **CODEOWNERS**. The whole point is a tight feedback loop that steers toward the right product while keeping it shippable.

Revision Notes

- Waterfall fails because requirements are discovered and change cost grows exponentially.
- Manifesto: individuals/interactions, working software, customer collaboration, responding to change — over the right-hand items.
- Scrum roles: PO (what/priority), Scrum Master (process/blockers), Dev team (how/build).
- Artifacts: product backlog → sprint backlog → increment.
- Ceremonies: planning → daily stand-up → review → retrospective.
- Kanban = continuous flow + WIP limits; no sprints.
- Points = relative size; velocity = empirical forecast; hours = punished guesses.
- INVEST for stories; Given/When/Then for acceptance criteria.
- DoR gates entry, DoD gates exit.
- Little's Law: cycle time = WIP / throughput → lower WIP to go faster.
- Ownership: strong / weak / collective; CODEOWNERS enforces review; bus factor = knowledge concentration.
- Conway's Law: system mirrors org communication; Inverse Conway to design teams.

Scrum vs Kanban

Aspect	Scrum	Kanban
Cadence	Fixed time-boxed sprints	Continuous flow
Commitment	Sprint goal + sprint backlog	No sprint commitment; pull as capacity frees
Roles	PO, Scrum Master, Dev team	No prescribed roles
Change mid-cycle	Discouraged (protect the sprint)	Allowed any time (re-prioritize backlog)
Core metric	Velocity	Cycle time / throughput
Board reset	Per sprint	Persistent
Best for	Plannable, batchable work	Interrupt-driven, unpredictable work

Waterfall vs Agile

Aspect	Waterfall	Agile
Planning	Big up-front, fixed	Continuous, adaptive
Delivery	One release at the end	Incremental, frequent
Feedback	Late (at the end)	Early and continuous
Change	Costly, resisted	Expected, embraced
Risk	Concentrated at integration/release	Spread and surfaced early
Docs	Comprehensive up front	Just enough, just in time

Ownership models

Model	Who edits	Pros	Cons
Strong	Only the owner touches their code	Deep expertise, clear accountability	Bottlenecks, low bus factor, blocked PRs
Weak	Owner exists; others contribute via review	Balance of expertise + flow; CODEOWNERS fit	Owner can still be a review bottleneck
Collective	Anyone may change anything	High bus factor, fast flow, no gatekeeping	Needs strong tests/standards; diffusion of responsibility

Practice Questions

1. Rewrite a vague ticket "make the app faster" into a proper INVEST user story with acceptance criteria.
2. Your "In Review" column always has 6 cards while "In Progress" has 1. What does the board tell you, and what do you change?
3. A manager wants to compare Team A's velocity (40) to Team B's (25) to judge productivity. Explain why this is invalid.
4. Draft a Definition of Ready for your team (at least 5 items).
5. Give a RACI for the decision "which state management approach will the app use."
6. Explain, using Little's Law, why cutting WIP from 8 to 4 could roughly halve cycle time.
7. Your PO is unavailable for two sprints. List three concrete consequences and one mitigation.

Coding Questions

Repurposed as practical process exercises.

1. **Break an epic into stories.** Epic: "User can manage their account." Produce at least four vertically-sliced user stories, each in "As a… I want… so that…" form, each with 2–3 Given/When/Then acceptance criteria, and assign relative points (1/2/3/5/8).
2. **Write a CODEOWNERS file** for a Flutter repo with `lib/features/auth`, `lib/features/feed`, `lib/design_system`, and `pubspec.yaml`, mapping each to a plausible team, with a catch-all default.
3. **Author a Definition of Done** tailored to a Flutter app that includes analyzer, formatting, widget tests, golden tests, and staged deployment. Then mark which items can be enforced automatically in CI vs which need a human.
4. **Design a Kanban board** (as a table or Mermaid) with columns and WIP limits for a 4-person team, and write one rule for what happens when a column hits its limit.
5. **Convert an hour estimate to points.** Given a team whose "3-point" story historically took ~2 days, forecast how many sprints 34 remaining points will take at a velocity of 18/sprint.

Mini Project

Run a simulated 2-week sprint for a small Flutter feature.

Goal: deliver "Offline caching of the product catalog" (the example story above) plus a couple of supporting stories.

Steps:

1. **Set up the board** (paper, Trello, or GitHub Projects) with columns: Ready, In Progress (WIP 2), In Review (WIP 2), In QA, Done. Write your DoR and DoD and pin them to the board.
2. **Backlog & refinement:** write 4–6 stories (catalog cache read, offline banner, pull-to-refresh graceful failure, cache invalidation, analytics event). Give each acceptance criteria and estimate in points via **planning poker** (everyone reveals a card simultaneously; discuss outliers; re-vote).
3. **Sprint planning:** pick a one-sentence sprint goal ("Users can browse the catalog offline"). Commit only to the points your (assumed) velocity supports.
4. **Simulate the days:** for five "days," hold a 2-minute stand-up, move cards, and inject one realistic blocker (e.g., "the cache library has a bug"). Practice swarming instead of starting new work when WIP is full.
5. **Trace one story ticket→production:** Ready → branch off main → implement behind a feature flag → open PR (CODEOWNERS auto-requests the catalog team) → review + CI (analyzer, tests, goldens) → merge → deploy to staging → PO verifies acceptance criteria → flag flipped on → Done. Cross-reference Version Control with Git and Testing Fundamentals & the Pyramid.
6. **Sprint review:** demo the working offline mode. Record one piece of feedback and add it to the backlog.
7. **Retrospective:** list what went well / what didn't / one action item with an owner. Measure your **velocity** (points actually Done) and note your longest **cycle time** — that's the bottleneck to attack next sprint.

Deliverables: the filled board, the story cards with criteria, the DoD/DoR, one CODEOWNERS file, and a short retro note. Success = a demoable increment and an honest velocity number you can plan the *next* sprint with.

Code Quality, Clean Code & Code Review

> Code quality is the discipline of keeping software cheap to change safely — through readable, testable, correct code and the review practices that protect it over time.

Introduction

Most of a software system's lifetime cost is spent *after* the first version ships: reading, understanding, changing, and fixing code that already exists. "Quality" is not a subjective aesthetic — it is a measurable property of how expensive the *next* change will be. This chapter treats quality as an **economic argument** and gives you the concrete practices — clean-code principles, refactoring, code review, and static analysis — that lower that cost.

Quality has four load-bearing dimensions:

Dimension	Question it answers	Failure symptom
Readability	Can a peer understand this in one pass?	"I have to run it to know what it does."
Maintainability	How cheap is the next change?	One feature touches ten files (shotgun surgery).
Testability	Can behavior be verified in isolation?	Untestable without a live network/DB.
Correctness	Does it do the right thing on all inputs?	Passes happy path, crashes on null/edge.

These are vendor-neutral engineering ideas; we illustrate them with Dart/Flutter, but the same reasoning applies in any language.

Why this concept exists

Code is written once and read hundreds of times. The dominant cost is **change over time**, and that cost compounds:

- A bug found in review costs minutes; the same bug in production costs hours plus a customer-trust tax.
- A confusing name is a small toll paid by *every* future reader, forever.
- Duplicated logic (WET — "write everything twice") means every change must be made in N places, and the one you forget becomes the outage.

The core insight: **software's value is not the code that exists today, but the code you can safely write tomorrow.** Clean code and review exist to keep the marginal cost of change flat rather than exponential. Uncontrolled complexity makes each change slower and riskier until the team's velocity collapses — the classic "big ball of mud."

This is why quality is a *business* concern, not a developer vanity project. A team that spends 20% up front on reviews and refactoring routinely spends far less than 100% later untangling a system nobody dares touch.

Real-world analogy

Think of a **professional kitchen**. Clean-as-you-go (the boy-scout rule: leave the station cleaner than you found it) keeps the kitchen operating at speed all night. Skip it, and by the dinner rush every surface is blocked, orders collide, and mistakes ship to tables.

- Meaningful names = clearly labelled containers; nobody confuses salt for sugar.
- Small functions = one station does one thing well.
- Code review = the head chef tasting a dish *before* it leaves the pass — cheap to fix now, expensive to recall from the customer.
- Linters/formatters = the health inspector's checklist, automated so humans discuss the food, not the floor.

A dirty kitchen can still serve dinner — for a while. Then it can't.

Problem Statement

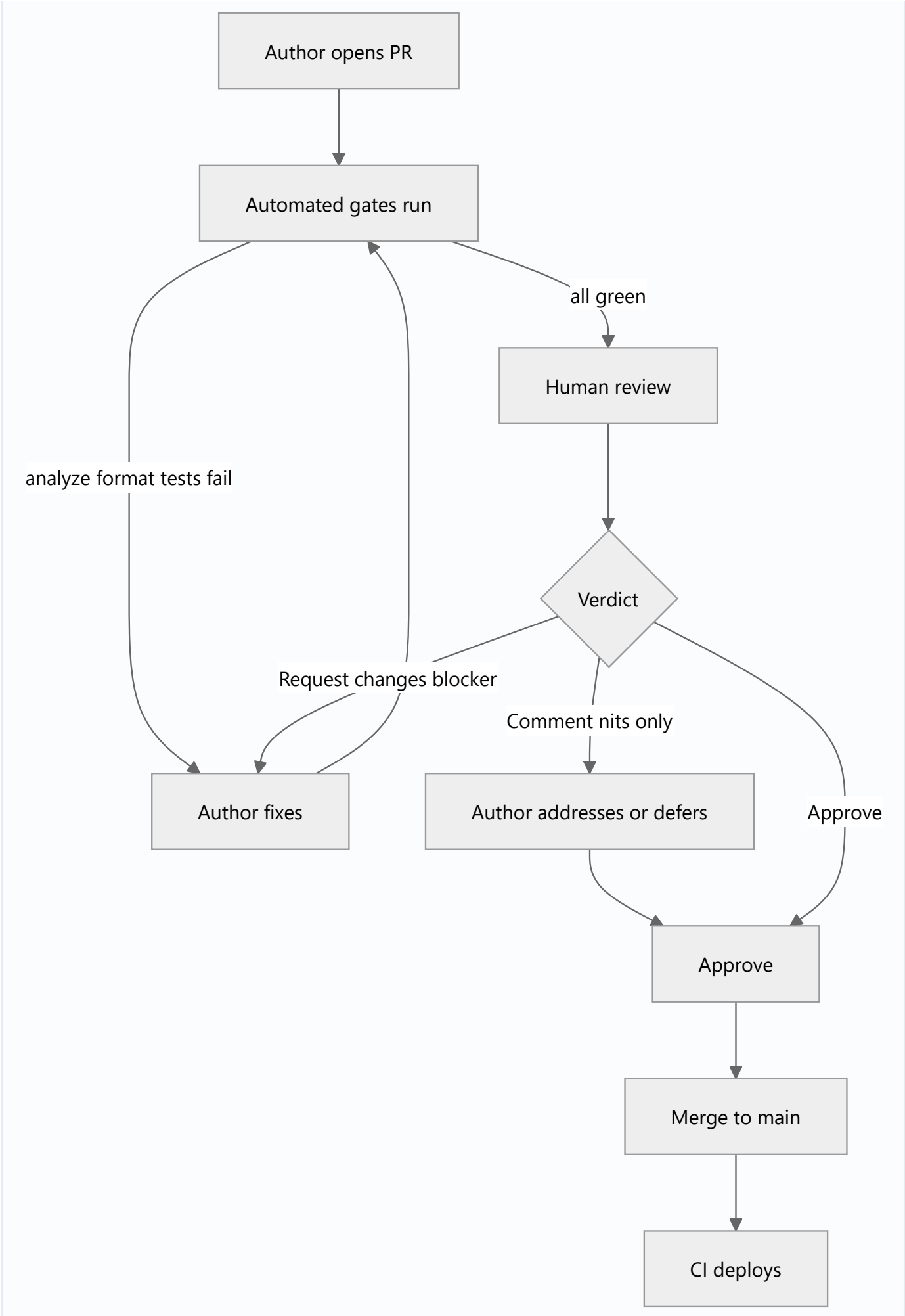
Without quality practices, predictable failures emerge:

1. **The code no one understands.** The original author left; the code is the only spec, and it's unreadable.
2. **The change that breaks three unrelated things.** Hidden coupling; no tests to catch it.
3. **The review that's a formality.** Rubber-stamped PRs let defects through; giant PRs make real review impossible.
4. **The bikeshed review.** Reviewers argue about brace style (a linter's job) while a real logic bug sails past.
5. **The frozen module.** So fragile that "don't touch it" becomes policy, and features route around it, accreting more mud.

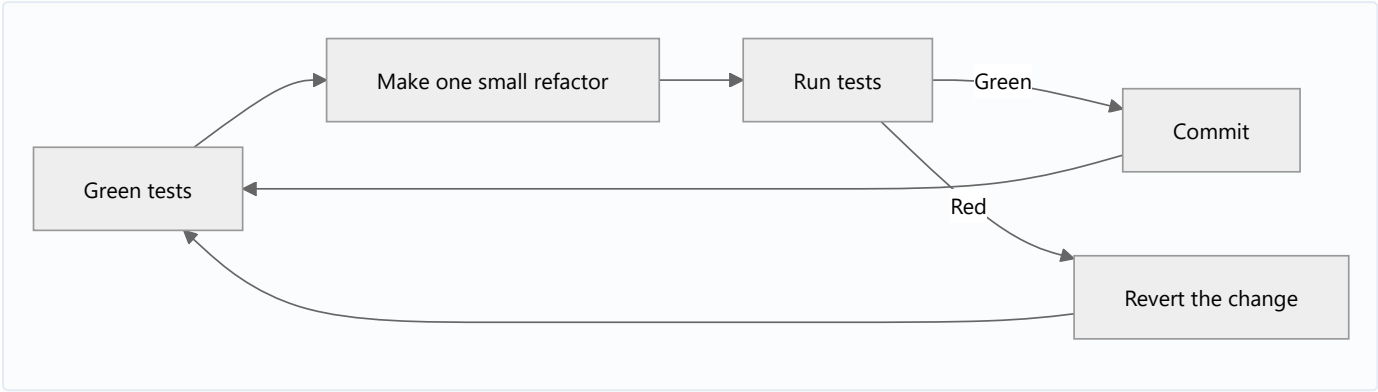
The goal of this chapter: make code that is *safe and cheap to change*, and build a review culture that catches defects early while staying kind, specific, and fast.

Internal Working

Two lifecycles power this practice. First, a pull request from open to merge:


Author opens PR
Automated gates run
all green
Human review
analyze format tests fail
Verdict
Request changes blocker
Comment nits only
Author fixes
Author addresses or defers
Approve
Approve
Merge to main
CI deploys

Second, the refactor-under-test loop — the only *safe* way to improve code without changing behavior:



The discipline: tests must be green *before* you start, each refactoring step is tiny and behavior-preserving, and you re-run tests after every step so any break is attributable to the last move.

Memory Representation

Repurposed: instead of RAM layout, this section is about **human working memory — cognitive load**, because that is the real "memory" clean code optimizes for.

Human working memory holds roughly 4–7 chunks at once. Every unclear name, deep nesting level, or long function forces the reader to hold more state in their head simultaneously. When you exceed the limit, comprehension fails and bugs slip in.

Clean code is a cognitive-load-reduction strategy:

Technique	What it removes from working memory
Meaningful names	The need to remember what `x`, `data2`, `tmp` mean.
Small functions	You reason about one function, not 200 lines at once.
Single level of abstraction	You aren't juggling "high-level intent" and "byte fiddling" in the same breath.
Guard clauses / shallow nesting	Fewer open mental brackets ("if inside if inside for...").
Making illegal states unrepresentable	You needn't remember "this is only valid when that flag is set."

A good name is a *cache entry*: it lets the reader recall a whole concept from one token. That is the sense in which this chapter has a "memory representation."

Compiler Behavior

Here the section genuinely applies: Dart ships a first-class **static analyzer** that enforces much of what we call quality, before any code runs.

- `dart analyze` performs static analysis: type errors, unused variables, dead code, null-safety violations, and lint-rule breaches. It uses the same analysis engine as the IDE.
- `dart format` deterministically reformats code, ending all whitespace/brace debates.
- **Lints** are configurable rules layered on top. `package:lints` (recommended set) and the stricter `package:flutter_lints` are common baselines.
- `analysis_options.yaml` is the control file: choose a lint set, toggle individual rules, and set the analyzer to fail on issues.

```yaml
# analysis_options.yaml
include: package:flutter_lints/flutter.yaml

analyzer:
  language:
    strict-casts: true
    strict-raw-types: true
  errors:
    # promote selected lints to hard errors (fail CI)
    unawaited_futures: error
    avoid_dynamic_calls: error
  exclude:
    - "**/*.g.dart"
    - "**/*.freezed.dart"

linter:
  rules:
    prefer_const_constructors: true
    prefer_final_locals: true
    always_declare_return_types: true
    require_trailing_commas: true
    avoid_print: true
```

The analyzer is a *compile-time quality gate*: it turns whole classes of review nits ("you forgot `final`", "unawaited future") into automated, non-negotiable feedback — freeing human reviewers to think about design and correctness. Custom lints (via `custom_lint` / `analyzer_plugin`) let a team encode its own rules, e.g. "no direct `DateTime.now()` in domain code."

Runtime Behavior

Repurposed — limited applicability: clean code is overwhelmingly a **compile-time and human concern**, with essentially **zero runtime cost**. Extracting a method, renaming a variable, or adding guard clauses does not change observable behavior — that is the definition of refactoring.

The one honest tension: **readability vs micro-performance**. Occasionally the clearest code is marginally slower — e.g., a readable `.map().where().toList()` chain allocates intermediate iterables versus one hand-rolled loop. The engineering rule:

1. Write it clearly first.
2. Measure with a profiler (see Performance).
3. Optimize only the proven hot path, and leave a comment explaining *why* the code is now less obvious.

99% of code is not hot. Sacrificing clarity for imagined speed is premature optimization — a code smell in itself.

Flutter Engine Behavior

Not applicable — because code quality and review are process/design practices that operate on source before it is ever compiled or rendered. The Flutter engine (Skia/Impeller rasterization, the C++ shell, platform channels) is entirely indifferent to whether your Dart is clean or filthy; it only ever sees compiled output. Widget-level *performance* practices belong in the rendering chapters, not here.

Dart VM Behavior

Not applicable — because refactoring is behavior-preserving by definition, so JIT/AOT compilation, isolate scheduling, and garbage collection see no semantic difference between "before" and "after" clean-up. Names and function boundaries are largely erased or inlined by the AOT compiler; the VM neither rewards nor penalizes readable code at runtime.

Examples

1. Bad naming → good naming (intent in the name).

```
// BEFORE — what is d? what is 7?
bool chk(User u, int d) => DateTime.now().difference(u.t).inDays > d;

// AFTER — the name is the documentation
bool isSubscriptionExpired(User user, {required int graceDays}) {
  final daysSinceRenewal = DateTime.now().difference(user.lastRenewedAt).inDays;
  return daysSinceRenewal > graceDays;
}
```

2. Long function → extract method + single level of abstraction.

```
// BEFORE — one function mixing validation, calculation, formatting
String buildInvoice(List<LineItem> items, double taxRate) {
  var subtotal = 0.0;
  for (final i in items) {
    if (i.quantity <= 0) throw ArgumentError('bad qty');
    subtotal += i.unitPrice * i.quantity;
  }
  final tax = subtotal * taxRate;
  final total = subtotal + tax;
  return 'Subtotal: \$${subtotal.toStringAsFixed(2)}\n'
      'Tax: \$${tax.toStringAsFixed(2)}\n'
      'Total: \$${total.toStringAsFixed(2)}';
}

// AFTER — each function reads at one level of abstraction
String buildInvoice(List<LineItem> items, double taxRate) {
  _validate(items);
  final subtotal = _subtotal(items);
  final tax = subtotal * taxRate;
  return _format(subtotal: subtotal, tax: tax);
}

void _validate(List<LineItem> items) {
  for (final item in items) {
    if (item.quantity <= 0) {
      throw ArgumentError.value(item.quantity, 'quantity', 'must be positive');
    }
  }
}

double _subtotal(List<LineItem> items) =>
    items.fold(0.0, (sum, i) => sum + i.unitPrice * i.quantity);

String _format({required double subtotal, required double tax}) =>
    'Subtotal: \$${subtotal.toStringAsFixed(2)}\n'
    'Tax: \$${tax.toStringAsFixed(2)}\n'
    'Total: \$${(subtotal + tax).toStringAsFixed(2)}';
```

3. Nested conditionals → guard clauses.

```
// BEFORE — arrow-shaped code, high cognitive load
String discountLabel(User? user) {
  if (user != null) {
    if (user.isActive) {
      if (user.isPremium) {
        return 'Premium discount';
      } else {
        return 'Standard discount';
      }
    } else {
      return 'Inactive';
    }
  } else {
    return 'Guest';
  }
}

// AFTER — flat, early returns, easy to scan
String discountLabel(User? user) {
  if (user == null) return 'Guest';
  if (!user.isActive) return 'Inactive';
  return user.isPremium ? 'Premium discount' : 'Standard discount';
}
```

4. Replace conditional with polymorphism + make illegal states unrepresentable.

```
// BEFORE — switch on a type tag, easy to forget a case
double area(String kind, double a, double b) {
  switch (kind) {
    case 'circle':
      return 3.14159 * a * a; // b ignored — illegal-state smell
    case 'rect':
      return a * b;
    default:
      throw ArgumentError('unknown shape');
  }
}

// AFTER — sealed class: compiler enforces exhaustive handling
sealed class Shape {
```

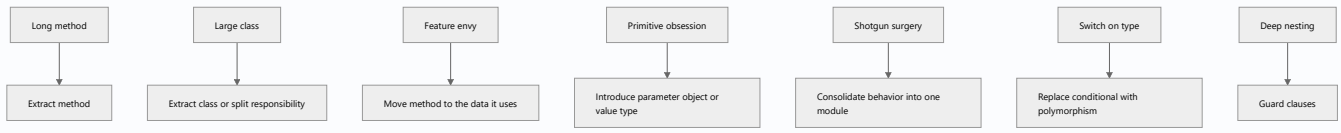
```
  double get area;
}

final class Circle extends Shape {
  Circle(this.radius);
  final double radius;
  @override
  double get area => 3.14159 * radius * radius;
}

final class Rectangle extends Shape {
  Rectangle({required this.width, required this.height});
  final double width;
  final double height;
  @override
  double get area => width * height;
}

// Adding a new Shape now forces every switch to handle it — fail fast, at compile time.
```

5. Introduce parameter object (primitive obsession → cohesive type).

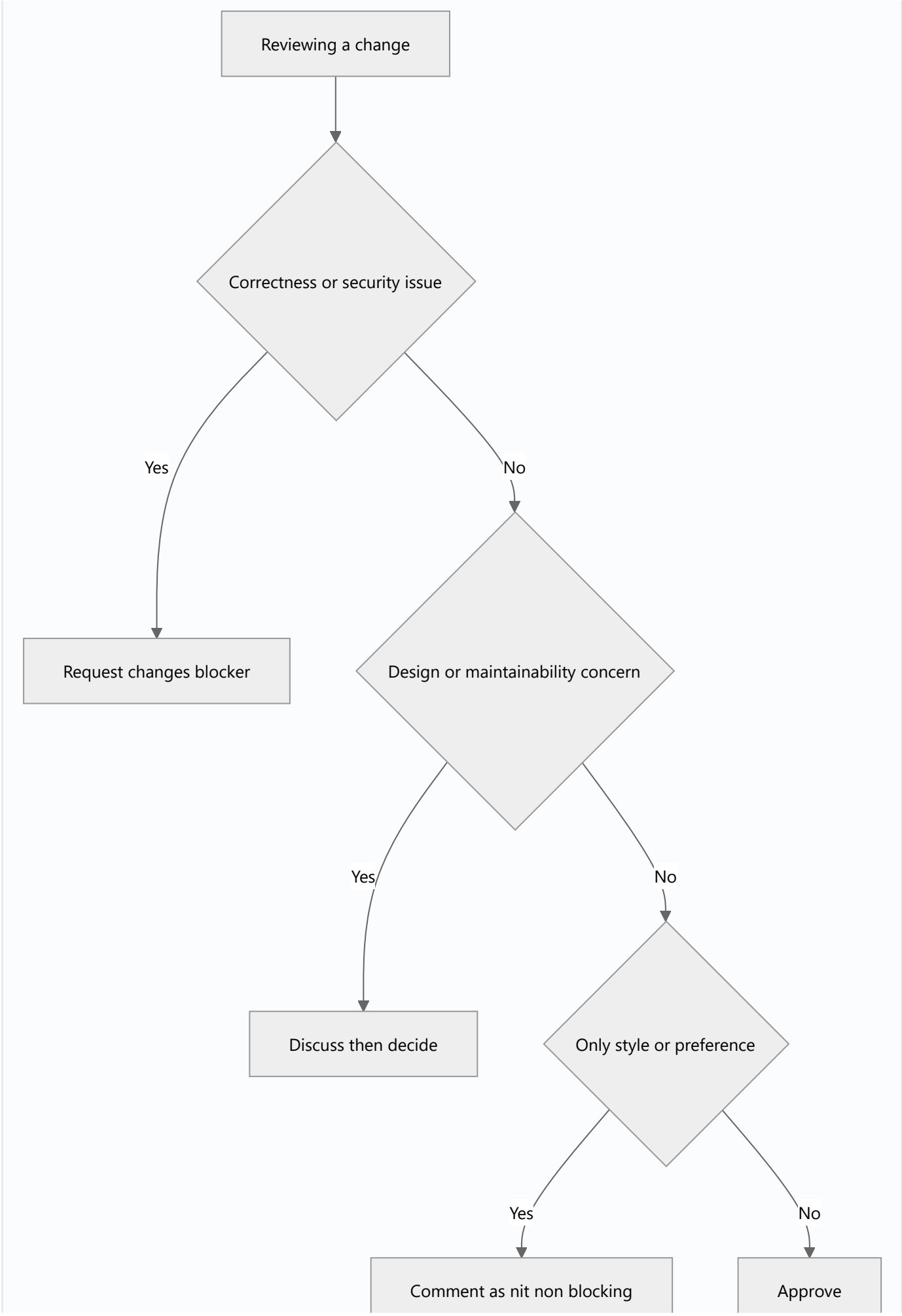
```
// BEFORE — a "data clump" of primitives passed everywhere
void book(String city, String country, String zip, double lat, double lng) {}

// AFTER — the concept has a name and can validate itself
class Location {
  Location({
    required this.city,
    required this.country,
    required this.zip,
    required this.lat,
    required this.lng,
  });
  final String city;
  final String country;
  final String zip;
  final double lat;
  final double lng;
}

void book(Location location) {}
```

Diagrams

Code smell → refactoring map:



Review verdict decision:


Reviewing a change
Correctness or security issue
Yes
No
Request changes blocker
Design or maintainability concern
Yes
No
Discuss then decide
Only style or preference
Yes
No
Comment as nit non blocking
Approve

Common Mistakes

Mistake	Why it hurts	Fix
Nitpicking style in review	Wastes human attention; demoralizes	Automate style with formatter+linter; humans review design
Rubber-stamping ("LGTM" unread)	Defects slip through; review becomes theatre	Require a specific comment or question per review; rotate reviewers
Giant PRs (1000+ lines)	Impossible to review well; reviewers skim	Keep PRs small and single-purpose; stack/split changes
Premature abstraction	Wrong abstraction is costlier than duplication	Follow "rule of three"; apply YAGNI; wait for real duplication
Refactoring without tests	Silent behavior changes	Add characterization tests first, then refactor
Comments that restate code	Rot into lies; add noise	Delete WHAT comments; keep WHY comments only
Mixing refactor + feature in one PR	Reviewer can't tell intent from accident	Separate "pure refactor" PRs from behavior changes
Blocking on personal preference	Slows the team; breeds resentment	Distinguish nit from blocker explicitly; defer to author on ties

Best Practices

- **Names carry intent.** A variable's name should make a comment unnecessary. Prefer `remainingRetries` over `n`.
- **Functions do one thing** and stay at one level of abstraction. If you need "and" to describe it, split it.
- **Guard clauses over nesting.** Handle edge cases first, return early, keep the happy path un-indented.
- **DRY, but not dogmatically.** Deduplicate *knowledge*, not incidental similarity. Two things that look alike but change for different reasons should stay separate.
- **KISS / YAGNI.** Build for today's requirement; don't speculate. Delete dead code — version control remembers it.
- **Boy-scout rule.** Every PR leaves the touched code slightly cleaner.
- **Comments explain WHY.** The code shows what; comments justify non-obvious decisions and link to tickets.
- **Fail fast.** Validate inputs at boundaries; throw or assert on impossible states rather than limping onward.

- **Make illegal states unrepresentable** with sealed classes, enums, and required fields (ties to SOLID).
- **Consistency over cleverness.** A codebase that reads the same everywhere is faster to work in than a collection of individually brilliant tricks.
- **Review small, review fast.** Aim for sub-day review latency; long-open PRs rot and force painful rebases.
- **Feedback is kind + specific + actionable.** "This function is confusing" is none of those; "Extracting the validation into `_validate` would make the intent clearer — wdyt?" is all three.

Performance

Repurposed to team-level metrics, because that is where "code quality" pays off measurably:

Metric	What it measures	Healthy signal
Review latency	Time from PR open to first review	Hours, not days
PR size	Lines changed per PR	Small; large PRs correlate with missed defects
Defect-escape rate	Bugs reaching production vs caught in review	Trending down
Change-failure rate	% of deploys causing incidents (DORA)	Low and stable
Lead time for change	Commit to production	Short — a proxy for maintainability

Quality practices move these numbers: automated gates cut review latency (humans stop policing style), small PRs cut defect-escape, and clean code cuts lead time because changes are safer.

The **readability-vs-performance tradeoff** is real but narrow: profile first, optimize the measured hot path only, and document why the optimized code is less obvious. Everywhere else, clarity wins because developer time is the scarce resource, not CPU cycles.

Advantages

- Lower total cost of ownership — cheaper, safer changes over the system's life.
- Faster onboarding; new engineers read intent, not archaeology.
- Fewer production defects; problems caught in review and by the analyzer.
- Shared understanding and bus-factor resilience across the team.
- Reviews spread knowledge and enforce standards without a central gatekeeper.

Disadvantages

- Up-front time cost; discipline is easy to skip under deadline pressure.
- Review can become a bottleneck or a politics arena if done poorly.

- "Clean" is partly subjective; teams need agreed conventions to avoid endless debate.
- Over-applied (gold-plating, premature abstraction) it wastes effort and adds complexity.
- Requires cultural buy-in; tools alone don't create quality.

Interview Questions

1. 🟢 **What does "code quality" actually mean, and why is it an economic argument?** Quality = readability, maintainability, testability, correctness. It's economic because most software cost is *change over time*; clean code keeps the marginal cost of the next change flat instead of exponential. You're optimizing for the code you'll safely write tomorrow.
2. 🟢 **DRY vs WET vs KISS vs YAGNI — define each.** DRY: don't duplicate *knowledge*. WET: the anti-pattern of writing the same thing repeatedly. KISS: keep it simple. YAGNI: don't build what you don't need yet. They balance each other — DRY without judgment causes premature abstraction, which YAGNI/KISS restrain.
3. 🟢 **What is refactoring, precisely?** A *behavior-preserving* change to code's structure to improve its internal quality. If observable behavior changes, it's not refactoring — it's a feature or a bug. Hence it requires test coverage to do safely.
4. 🟡 **Walk me through how you review a PR.** First I read the description and linked ticket to understand *intent*. I check CI/analyzer are green so I'm not reviewing style manually. I read tests to see what behavior is claimed, then the diff for correctness and edge cases (null, empty, error paths), then design/maintainability. I flag blockers (correctness, security) clearly, mark preferences as non-blocking "nits", ask questions rather than dictate, and keep feedback kind, specific, actionable. I approve if there's no blocker, even with open nits I trust the author to weigh.
5. 🟡 **How do you give feedback that lands well?** Kind + specific + actionable. Critique the code, not the person; explain the *why*; suggest a concrete alternative; distinguish must-fix from preference; use "we/this" not "you". Praise good bits too — review isn't only fault-finding.
6. 🟡 **Name four code smells and their refactorings.** Long method → extract method; large class → extract class; feature envy → move method to the data it uses; primitive obsession → introduce a value type/parameter object; switch-on-type → replace conditional with polymorphism.
7. 🟡 **How does `dart analyze` differ from `dart format`, and what's `analysis_options.yaml` for?** `analyze` is static analysis — finds type errors, dead code, lint violations. `format` deterministically reformats whitespace. `analysis_options.yaml` configures the analyzer: which lint set, individual rule toggles, promoting lints to errors, and excludes. Together they automate the mechanical parts of review.

8. 🔴 **What's the danger of DRY, and how do you decide when *not* to deduplicate?** The wrong abstraction is more expensive than duplication because it couples things that change independently. Two code blocks that look identical but change for *different reasons* should stay separate (Single Responsibility). Rule of three: tolerate duplication until you've seen it three times and understand the shared axis of change.
9. 🔴 **How do you refactor a legacy module with no tests?** Write *characterization tests* first — tests that pin the current behavior, warts included — using the module as an oracle. Once green, refactor in tiny steps, re-running tests each step. Introduce seams (dependency injection) to make it testable. See Testing.
10. 🔴 **"Make illegal states unrepresentable" — what does it mean in Dart?** Design types so invalid combinations can't be constructed: `required` fields, non-nullable types, enums/sealed classes instead of stringly-typed tags, and factory constructors that validate. The compiler then enforces invariants, moving whole bug classes from runtime to compile time — fail fast, statically.
11. 🟡 **How do you keep code review from becoming a bottleneck?** Small PRs, SLA on review latency, automate style/lint, distinguish nits from blockers, allow "approve with comments", rotate reviewers, and let authors merge after addressing blockers without a second full round for trivia.
12. 🔴 **When does clean code conflict with performance, and how do you resolve it?** Rarely, and only in hot paths (e.g., iterator chains allocating vs a manual loop). Resolve by: write clear first, profile, optimize only the measured hot spot, and document why the optimized code is less obvious. Never trade clarity for unmeasured speed.

Senior Engineer Tips

- Review the **tests first** — they tell you what the author *intended*; the implementation tells you what they *did*.
- Separate refactor PRs from feature PRs. A diff that both moves code and changes behavior is un-reviewable.
- Prefix non-blocking comments with `nit:` so juniors know what's optional. It changes the whole tone.
- If you can't understand the code in one pass, that *is* the review finding — say so, kindly.
- Automate every rule you find yourself repeating in reviews into a lint. Say it once to a human, then to the linter forever.
- Leave the campsite cleaner, but scope the cleanup to what you touched; don't sneak a rewrite into a bugfix.
- Approve generously with nits; block only on correctness, security, and genuine design harm. Trust your teammates.

Architect Perspective

At scale, the challenge is **enforcing standards without becoming the bottleneck**:

- **Quality gates in CI**, not in a person's head: `dart analyze` with warnings-as-errors, `dart format --set-exit-if-changed`, coverage thresholds, and required green checks before merge. The pipeline is the tireless first reviewer.
- **Codified standards.** A shared `analysis_options.yaml` (published as an internal lint package) means "the style" is a dependency, not a wiki page nobody reads. Custom lints encode architecture rules ("domain layer must not import Flutter").
- **Distribute review authority.** CODEOWNERS routes PRs to the right eyes; you scale by growing reviewers, not by funnelling everything through architects.
- **Guardrails over gates where possible.** Prefer analyzers and templates that make the right thing easy over manual approvals that make everything slow.
- **Measure the system, not the people.** Track DORA metrics and defect-escape rate; use them to find process friction, never to rank individuals.
- **Manage technical debt explicitly.** A visible debt backlog and a "boy-scout" budget per sprint keep quality from being perpetually deprioritized. Tie module boundaries to SOLID and Design Patterns so the architecture itself resists rot.

Summary

Code quality is the economics of change: readable, maintainable, testable, correct code keeps the next change cheap and safe. Clean-code principles (meaningful names, small single-purpose functions, single level of abstraction, guard clauses, WHY-comments, DRY/KISS/YAGNI, boy-scout rule) reduce human cognitive load. Code smells signal where to refactor — behavior-preserving structural improvement done safely under test coverage. Code review is the human quality gate: kind, specific, actionable feedback; small PRs; low latency; nits distinguished from blockers; boring parts automated. The Dart analyzer, formatter, and `analysis_options.yaml` enforce mechanical quality at compile time so humans focus on design and correctness. It ties directly to SOLID, testing, and version control.

Revision Notes

- Quality = readability + maintainability + testability + correctness; justification = cost of change over time.
- Refactoring = **behavior-preserving**; requires tests; small steps, re-run tests each step.
- Smell → refactoring: long method→extract; large class→extract class; feature envy→move method; primitive obsession→value type; switch-on-type→polymorphism; deep nesting→guard clauses.

- Review verdict: blocker (correctness/security) → request changes; preference → non-blocking nit; else approve.
- `dart analyze` = static analysis; `dart format` = formatting; `analysis_options.yaml` = config, includes lint set + rule toggles + errors + exclude.
- Fail fast; make illegal states unrepresentable; consistency over cleverness.
- Compiler Behavior IS applicable (the analyzer). Flutter Engine / Dart VM: not applicable — refactoring is behavior-preserving. Memory→cognitive load; Runtime/Performance→repurposed.

Practice Questions

1. Explain to a non-technical manager why spending time on code review saves money. Use the cost-of-change argument.
2. Give one example each of a WHY comment (keep) and a WHAT comment (delete).
3. You see the same 5-line block in three files. Walk through your decision on whether to DRY it up.
4. A teammate's PR is 1,400 lines mixing a refactor and a feature. What do you ask for, and why?
5. List the four dimensions of quality and give a Dart symptom of each being violated.
6. Rewrite a 4-level-nested `if` (of your choosing) using guard clauses.
7. When is it correct to *keep* duplicated code? Justify with the "axis of change" idea.

Coding Questions

Q1 — Guard-clause refactor. Given:

```
double? price(Map<String, dynamic> json) {
  if (json.containsKey('item')) {
    final item = json['item'];
    if (item is Map) {
      if (item.containsKey('price')) {
        final p = item['price'];
        if (p is num) {
          return p.toDouble();
        }
      }
    }
  }
  return null;
}
```

Acceptance criteria: null-safe; max one level of nesting; behavior identical for all inputs; lint-clean under `flutter_lints`; guard clauses used.

Q2 — Extract method + name for intent. Refactor a 40-line `buildInvoice`-style function so each function is under ~10 lines and reads at a single level of abstraction. Acceptance: no function does more than one thing; all names reveal intent; a unit test asserting the output exists and passes before and after (behavior-preserving).

Q3 — Replace conditional with polymorphism. Convert a `switch (kind)` over `'circle' | 'rect' | 'triangle'` into a `sealed class Shape` hierarchy. Acceptance: adding a fourth shape causes a *compile-time* error anywhere a shape is exhaustively handled; no `default`/`else` fallthrough for known shapes.

Q4 — Write an `analysis_options.yaml`. Produce a config that: includes `package:flutter_lints/flutter.yaml`; enables `prefer_final_locals`, `always_declare_return_types`, `require_trailing_commas`, `avoid_print`; promotes `unawaited_futures` to an error; enables `strict-casts`; excludes generated files (`*.g.dart`, `*.freezed.dart`). Acceptance: `dart analyze` runs without config errors; a stray `print()` produces a warning; an unawaited future fails the analysis.

Mini Project

Refactor a deliberately messy Dart file to clean, under tests.

Starting point — a "cart total" utility with multiple smells (poor names, long method, deep nesting, primitive obsession, magic numbers):

```
// messy: cart.dart  (DO NOT SHIP)
double calc(List l, String c, bool m) {
  double t = 0;
  for (var i = 0; i < l.length; i++) {
    if (l[i]['q'] > 0) {
      if (l[i]['p'] > 0) {
        t = t + l[i]['q'] * l[i]['p'];
      }
    }
  }
  if (c == 'SAVE10') {
    t = t - t * 0.1;
  }
  if (m == true) {
    t = t + 5;
  }
  return t;
}
```

Steps (do them in this order):

1. **Pin behavior first.** Write characterization tests covering: empty cart, valid items, zero/negative quantities, the `SAVE10` coupon, and the membership-shipping flag. Get them green against the messy code. See Testing.
2. **Introduce types** (primitive obsession → value objects): `CartItem` with `quantity` / `unitPrice`, a `Coupon` sealed/enum type, and a `ShippingOption`. Make illegal states unrepresentable.
3. **Extract methods** at one level of abstraction: `_subtotal`, `_applyCoupon`, `_shipping`.
4. **Flatten nesting** with guard clauses; name everything for intent; replace magic numbers (`0.1`, `5`) with named constants.
5. **Re-run tests after every step** — they must stay green (behavior-preserving).
6. **Self-review the diff** as if it were a PR: is any function doing two things? Any name still cryptic? Any nit a linter should own?

Target shape:

```
class CartItem {
  CartItem({required this.quantity, required this.unitPrice});
  final int quantity;
  final double unitPrice;
  double get lineTotal => quantity > 0 && unitPrice > 0 ? quantity * unitPrice : 0;
}

enum Coupon { none, save10 }

class Cart {
  Cart(this.items);
  final List<CartItem> items;
  static const _shippingFee = 5.0;
  static const _save10Rate = 0.10;

  double total({Coupon coupon = Coupon.none, bool memberShipping = false}) {
    final subtotal = _subtotal();
    final discounted = _applyCoupon(subtotal, coupon);
    return discounted + _shipping(memberShipping);
  }

  double _subtotal() => items.fold(0.0, (sum, item) => sum + item.lineTotal);

  double _applyCoupon(double amount, Coupon coupon) =>
      coupon == Coupon.save10 ? amount * (1 - _save10Rate) : amount;

  double _shipping(bool memberShipping) => memberShipping ? _shippingFee : 0.0;
}
```

Acceptance criteria: all characterization tests still pass; zero `dart analyze` issues under `flutter_lints`; no function exceeds ~10 lines; no magic numbers; no `dynamic` / `Map` primitive obsession remains; the change is committed as a **pure refactor** (no behavior change) with a message that says so. Optionally record the before/after in version control as two separate commits.

Documentation, RFCs & Design Docs

> Documentation is the deliberate act of moving knowledge out of individual heads and into a durable, discoverable, shared artifact — and RFCs/design docs are that act applied *before* the code exists, so that thinking is reviewed instead of just merged.

Introduction

Code tells you *what* the system does right now. It almost never tells you *why* it does it that way, what was tried and rejected, or what will break if you change it. That "why" lives in people's heads — until those people go on vacation, switch teams, or leave.

Documentation is the engineering discipline of externalizing that knowledge. It spans a spectrum: from a one-line `///` comment above a function, to a README, to a multi-page **RFC** (Request For Comments) that proposes a design before anyone writes code, to an **ADR** (Architecture Decision Record) that freezes a single decision forever.

This chapter is about the *writing craft* — how to write docs people actually read, how to structure a design doc that survives review, and why the act of writing is itself a thinking tool. It is deliberately vendor-neutral: the same practices apply whether you ship Flutter, a backend, or firmware.

> This topic overlaps module 58's material on decision records. Here we focus on **writing** — structure, prose, audience. For the *decision-making* side (how to weigh tradeoffs, decision frameworks), see Decision Frameworks & Tradeoffs.

Why this concept exists

Engineers under-document for predictable, human reasons — and each one has a team-level cost:

Why engineers skip docs	What it costs the team
"The code is self-documenting"	Code shows *what*, never *why*; intent is lost
"I'll remember"	You won't; and you're now the single point of failure (**bus factor = 1**)
"Writing is slower than coding"	True locally, false globally — every future reader re-derives what you knew
"It'll be stale in a month"	Stale-but-dated beats absent; and staleness is a maintenance problem, not a reason to never write
"Nobody reads docs"	Nobody reads *bad* docs; audience-targeted docs get read

The three concrete failure modes documentation prevents:

1. **Bus factor.** If knowledge lives only in one person, the team stalls the moment that person is unavailable. Docs raise the bus factor.
2. **Onboarding cost.** A new hire with good docs is productive in days; without them, they interrupt senior engineers for weeks.
3. **Re-litigated decisions.** Without a record of *why* a decision was made, teams re-argue it every 6 months. An ADR ends the argument: "we decided this, here's the context, here are the consequences."

RFCs exist for a fourth reason: **catching design mistakes when they are cheap.** A flaw found in a doc costs an hour to fix; the same flaw found in production costs a rewrite.

Real-world analogy

Think of a hospital patient chart. No single nurse or doctor holds the full history in their head — the chart does. Shift changes happen constantly, yet care is continuous because the *chart* is the memory, not any individual. A verbal handoff ("the patient in bed 3 is fine") is a stale comment; the written chart is documentation.

A **design doc** is the pre-surgery plan the surgical team reviews together *before* the first incision — goals, risks, contingencies, what happens if things go wrong. You would not want a surgeon who "figures it out as they go." An **ADR** is the immutable line in the record: "administered drug X at 14:32 because of Y" — you never edit it later; if the situation changes you add a *new* entry.

Problem Statement

A team of six ships a Flutter app. The engineer who built the offline-sync layer leaves. Six months later:

- A bug appears in conflict resolution. Nobody knows whether "last-write-wins" was a deliberate choice or an accident.
- A PM asks to add multi-device sync. The team spends a full sprint re-discovering constraints the original engineer already knew.
- A new hire spends two weeks reading source code to understand what a half-page doc could have explained in twenty minutes.

The knowledge existed — it just never left one person's head. The problem this chapter solves: **how to reliably externalize engineering knowledge in forms that match how it will be consumed.**

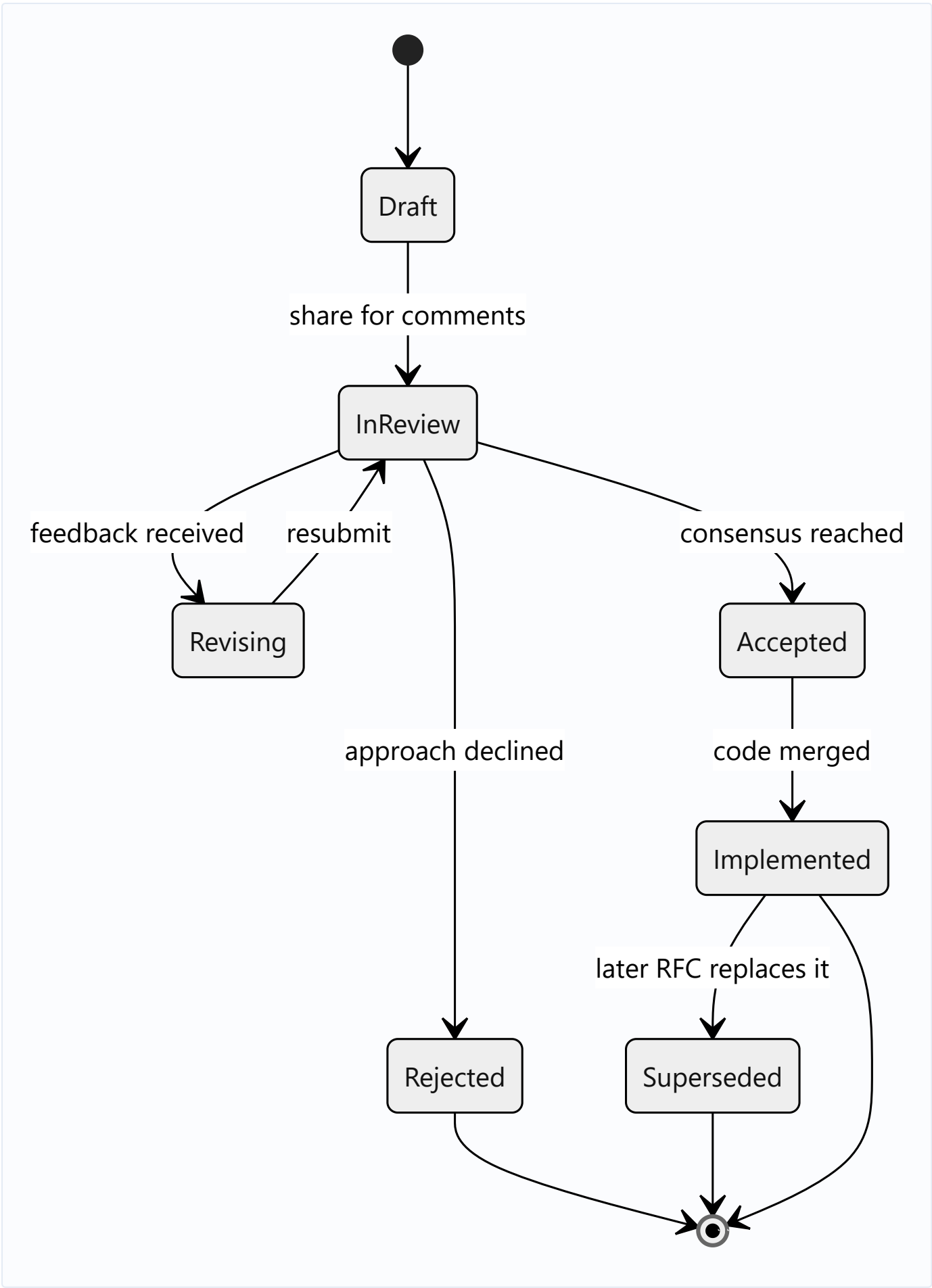
Internal Working

Documentation is not one thing; it is a *spectrum*, each type serving a different audience and lifespan. And the RFC has a defined lifecycle from draft to accepted.

The documentation spectrum:

Doc type	Lives where	Answers	Audience	Lifespan
Code comment (`//`)	Inline in code	Why *this line* is weird	The next code reader	Same as the line
Doc comment (`///`)	Above a symbol	How to *use* this API	API consumers	Same as the API
README	Repo root	How to build/run this	New contributors	Life of the repo
API docs (dartdoc)	Generated site	Full API surface	External + internal users	Regenerated each release
Runbook	Wiki / repo	How to operate this in prod	On-call engineers	Life of the service
Wiki	Confluence / repo	Broad "how things work"	Whole team	Long, decays fastest
Changelog	`CHANGELOG.md`	What changed per version	Upgraders	Append-only, forever
RFC / design doc	Repo `docs/`	*Why* we will build it this way	Reviewers, future you	Frozen once accepted
ADR	`docs/adr/`	One decision + consequences	Future maintainers	Immutable, forever

The RFC lifecycle — the path a design proposal travels:


Draft
share for comments
InReview
feedback received
resubmit
consensus reached
Revising
Accepted
approach declined
code merged
Implemented
later RFC replaces it
Rejected
Superseded

The key insight: an RFC is **socialized** — deliberately circulated — during `InReview`. The comments *are* the value. A doc nobody comments on either was perfect or nobody read it, and it's rarely the former.

Memory Representation

Repurposed: institutional / organizational memory.

In a normal chapter this section covers bytes in RAM. Here we repurpose it, because documentation *is* a form of memory — just at the organizational layer instead of the machine layer.

A running program keeps state in memory so it doesn't have to recompute everything. A team keeps *decisions and rationale* in documentation so it doesn't have to re-derive them. The analogy is exact:

Machine memory	Organizational memory
RAM (fast, volatile)	What's in engineers' heads today
Disk (durable)	Docs, ADRs, RFCs in the repo
Cache eviction	An engineer leaving = memory lost
Persisting to disk	Writing it down before it's lost
Cache miss → recompute	Undocumented decision → re-litigated

Docs are the team's **write-to-disk** operation. An ADR is a durable record that survives the "process" (the engineer) exiting. This is why ADRs are *immutable*: you don't corrupt old memory, you append new records — exactly like an append-only log.

Compiler Behavior

*Here, partly applicable — via **dartdoc**.*

Dart ships a documentation generator, `dartdoc`. It is not the compiler proper, but it is a tool in the toolchain that reads your source the way a compiler does and produces an artifact from it.

What dartdoc does:

- Parses your `.dart` files and extracts every **doc comment** — comments written with `///` (or `/** */`) directly above a declaration.
- Treats the doc-comment text as **Markdown**.
- Resolves **symbol references** written in square brackets — `[SomeClass]`, `[myMethod]` — into hyperlinks to that symbol's page.
- Generates a static HTML site documenting your public API surface (or the whole thing with `--include-dependencies`-style flags).

```
dart doc .          # generates docs into doc/api/
```

Two consequences worth internalizing:

1. **Doc comments are compiled artifacts, not throwaway text.** A broken `[symbolRef]` produces a dartdoc warning, just like a type error produces a compiler warning. The `public_member_api_docs` lint even *fails your build* if public members lack docs, when enabled.
2. **`///` is semantically distinct from `//`.** The analyzer associates `///` with the following declaration; `//` is ignored by dartdoc. Using `//` for API documentation means it never reaches the generated site.

So: the doc generator turns your `///` comments into the same API reference site you use when you read Flutter's own documentation. Your comments and Flutter's are processed by the identical tool.

Runtime Behavior

Repurposed: how docs are consumed and kept fresh.

Documentation has no runtime in the program-execution sense — it never runs. But it does have a *consumption* lifecycle that behaves like a runtime for the team:

- **Read path:** a developer hits a question → searches the wiki/README/API docs → finds (or fails to find) an answer. *Time-to-answer* is the latency metric.
- **Refresh path:** code changes → the doc that described it is now wrong → someone must update it. If nothing forces this, docs *decay*. Decay is the documentation equivalent of a memory leak: unnoticed until it causes a crash (a wrong doc that misleads someone).
- **Invalidation:** the dangerous state is a doc that is *confidently wrong*. A missing doc makes you ask a human; a wrong doc makes you ship a bug. This is why "delete stale docs" is a legitimate maintenance action.

The mechanism that keeps docs fresh: **co-location + review**. Docs living in the repo (docs-as-code) get changed in the same PR as the code, and the reviewer catches the mismatch. Docs in a separate wiki drift because nothing links their fate to the code's.

Flutter Engine Behavior (if applicable)

Not applicable — because documentation is an authoring-time and human-process artifact. The Flutter engine (Skia/Impeller rasterization, the C++ shell, platform channels) never reads, executes, or is affected by your prose or design docs. Doc comments are stripped from compiled output and have zero presence in the running engine.

Dart VM Behavior (if applicable)

Not applicable — because doc comments and design docs are discarded before execution. The Dart VM (JIT/AOT compilation, garbage collection, isolates) operates on compiled kernel/machine code from which comments have been removed. Documentation influences the *humans* who write the code the VM runs, never the VM itself.

Examples

1. A Dart doc comment with `///` and symbol references

```
/// A store that persists key-value pairs locally and syncs them when online.
///
/// Values are written immediately to the local cache and queued for upload.
/// Use [flush] to force a sync, or rely on the automatic retry described in
/// [SyncPolicy]. Reads always return the local value; see [get].
///
/// ```dart
/// final store = OfflineStore(policy: SyncPolicy.eager);
/// await store.put('theme', 'dark');
/// ```
///
/// Throws a [StateError] if used after [dispose] has been called.
class OfflineStore {
  /// Writes [value] under [key] to the local cache and queues an upload.
  ///
  /// Returns once the value is durably in the local cache — **not** once it
  /// has synced to the server. To await the server round-trip, call [flush].
  Future<void> put(String key, String value) async {
    // ...
  }

  /// Forces any queued writes to sync now, completing when the server acks.
  ///
  /// Prefer this over polling; it respects the backoff in [SyncPolicy].
  Future<void> flush() async {
    // ...
  }

  /// Reads the locally cached value for [key], or `null` if absent.
  String? get(String key) => throw UnimplementedError();
```

```
  /// Releases resources. Calling any method afterward throws [StateError].
  void dispose() {}
}
```

Note the craft: the first sentence is a **standalone summary** (dartdoc uses it as the tooltip/list description), `[symbol]` refs link to other declarations, an example is fenced, and each comment says *why/contract* (`not once it has synced`) rather than restating the signature.

2. A filled-in mini RFC

```
# RFC 0007: Offline Write Queue for the Orders Screen

Status: Accepted   Author: A. Rao   Date: 2026-06-30   Reviewers: 3

## Context / Problem
Field reps create orders in areas with no connectivity. Today a failed
network call drops the order and the rep must retype it. We lose ~30
orders/week to this.

## Goals
- Orders created offline are never lost.
- Reps see clear "pending sync" state per order.

## Non-Goals
- Offline *edits* to server-created orders (future RFC).
- Conflict resolution beyond last-write-wins.

## Proposed Design
Wrap order creation in a local durable queue (Drift table `outbox`).
A background worker drains the queue with exponential backoff when
connectivity returns. UI reads reflect queued state via a stream.

## Alternatives Considered
- **In-memory queue:** rejected — lost on app kill.
- **Full CRDT sync:** rejected — over-engineered for create-only, non-Goal.

## Tradeoffs
Gain durability + simplicity; accept last-write-wins can silently drop a
concurrent server change (acceptable: orders are append-only).
```

```
## Risks

- Queue never drains if backoff has a bug → add a max-age alert.
- Duplicate submission on retry → server dedupes on client-generated UUID.


## Rollout Plan

Behind flag `offline_orders`. 5% → 50% → 100% over two weeks, watching
the duplicate-order metric.
```

3. A one-page ADR (context → decision → consequences)

```
# ADR 0012: Use client-generated UUIDs for order IDs


Status: Accepted   Date: 2026-06-30


## Context

The offline write queue (RFC 0007) may retry a submission after a network
timeout, risking duplicate orders. IDs were previously assigned by the server.


## Decision

Clients generate a v4 UUID for each order at creation time. The server treats
the ID as idempotency key and rejects duplicates.


## Consequences

- (+) Retries are naturally idempotent; no duplicates.
- (+) Orders have a stable ID before ever reaching the server.
- (-) Client trust: a malicious client could forge IDs — mitigated by
      server-side ownership checks.
- (-) UUIDs are larger than sequential ints in the DB index.
```

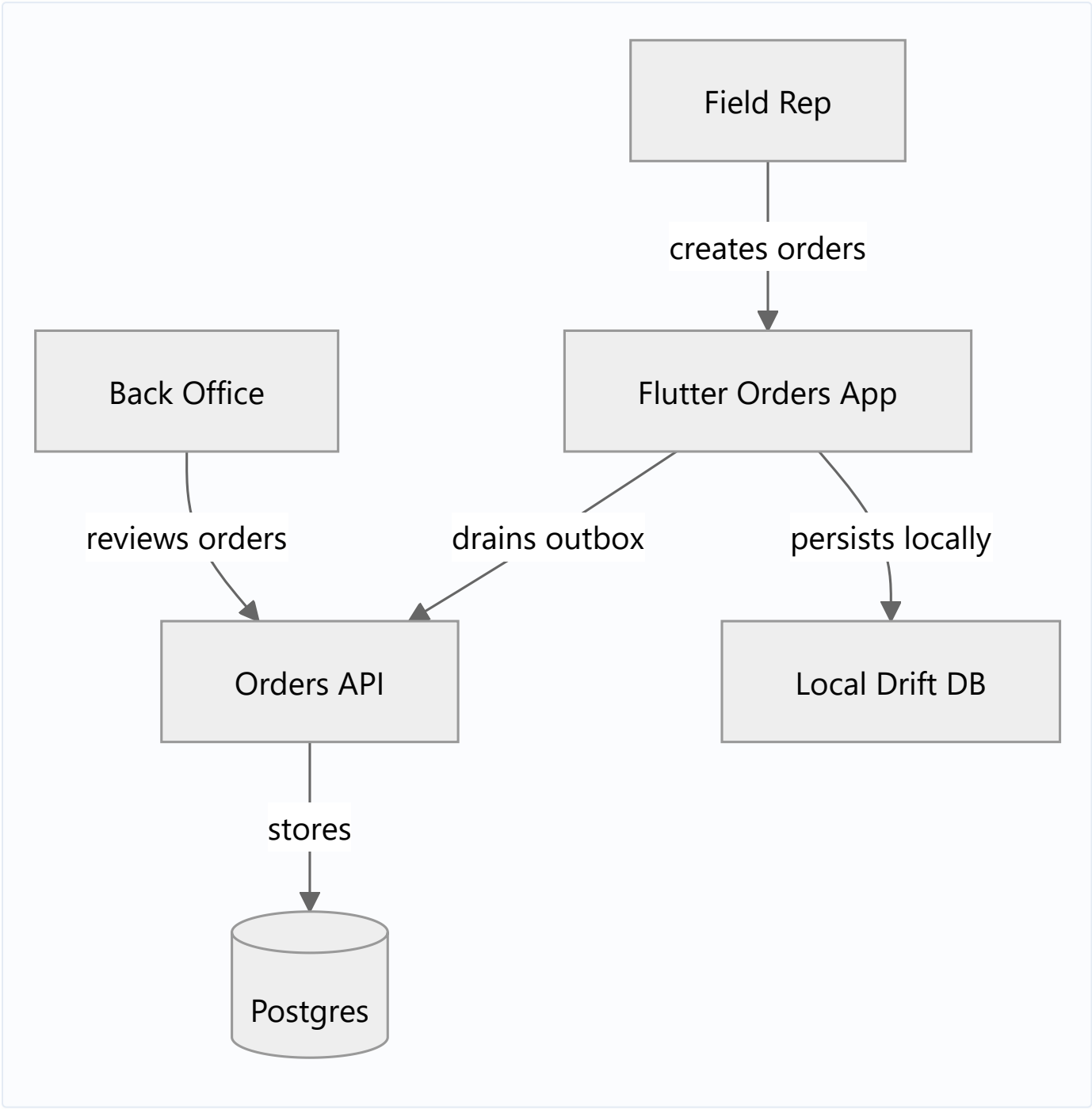
Diagrams

When a diagram beats prose: any time the relationships between more than ~three things matter more than the things themselves — sequences, hierarchies, data flow. Prose is linear; systems are not.

C4 model levels (progressive zoom): **Context** (system + external actors) → **Container** (apps/services/DBs) → **Component** (parts inside one container) → **Code** (classes; rarely worth drawing). Pick the level that answers the reader's question; don't draw all four by default.

Sequence vs component: use a *sequence* diagram to show *ordering over time* (who calls whom, in what order); use a *component* diagram to show *static structure* (what talks to what). "How does login work?" → sequence. "What are the pieces?" → component.

C4 Level-1 context diagram for the offline-orders feature:



Common Mistakes

Mistake	Why it hurts	The fix
Doc describes **what**, not **why**	`// increments i` adds nothing; the code already says that	Document intent/contract: `// retry once — the gateway 500s on cold start`
Stale docs left in place	A confidently-wrong doc ships a bug	Update docs in the *same PR* as the code; delete docs you can't maintain
Design doc written **after** the fact	It becomes a rubber-stamp, not a review; mistakes are already shipped	Write the RFC *before* coding, when changing course is cheap
No non-goals	Scope creeps; reviewers argue about things you never intended to build	Always include a Non-Goals section — it's as important as Goals
Doc written for the **wrong audience**	Too much detail for a PM, too little for an implementer	State the audience at the top; write to them
Symbol refs as plain text in `///`	No hyperlink in generated docs	Use `[SymbolName]` so dartdoc links it
Everything in a **wiki** far from code	Drifts instantly; nobody updates it	Docs-as-code: keep them in the repo next to what they describe

Best Practices

- **Write the first sentence to stand alone.** dartdoc, search results, and tooltips show only that sentence.
- **Docs-as-code.** Keep docs in the repo, in Markdown, reviewed via PR. Their fate is tied to the code's.
- **One decision per ADR, and never edit an accepted one.** Supersede with a new ADR instead.
- **Goals *and* Non-Goals.** Bounding the scope is half the value of a design doc.
- **Socialize RFCs early.** Share a rough draft for comments; the feedback is the point.
- **Say why, not what.** The code is the source of truth for *what*; docs own *why*.
- **Assign an owner.** A doc with no owner is a doc that will go stale.
- **Prefer a diagram** when relationships dominate; keep node labels short and plain.
- **Enable the `public_member_api_docs` lint** for published packages to make undocumented public APIs a build issue.

Performance

Repurposed: documentation's performance is measured in team-level latencies, not milliseconds.

Metric	What it measures	Good docs move it
Onboarding time	Days until a new hire ships confidently	Weeks → days
Time-to-find-an-answer	Latency from question to answer	Interrupt-a-senior → self-serve in minutes
Decision-reversal rate	How often decisions are re-litigated	ADRs drop it toward zero
Bus factor	People who must be present for the team to function	Raises it above 1
Incident MTTR	Time to resolve a prod incident	A runbook cuts it dramatically

If you cannot measure a doc's effect on one of these, ask whether it should exist.

Advantages

- Raises the bus factor; knowledge survives people leaving.
- Slashes onboarding cost — the biggest hidden tax on growing teams.
- Ends re-litigation of settled decisions (ADRs).
- Catches design flaws while they're cheap to fix (RFCs).
- Writing forces clarity — you cannot write a muddy design doc for a design you don't actually understand.
- Enables async, distributed collaboration — reviewers comment on their own schedule.

Disadvantages

- Costs time up front; the payoff is deferred and diffuse.
- Can go stale and mislead if not maintained — sometimes worse than nothing.
- Over-documentation buries the important docs under trivial ones.
- Requires cultural buy-in; a lone documenter on an indifferent team burns out.
- Bad docs (what-not-why, wrong audience) can create false confidence.

Interview Questions

🟢 **1. What's the difference between `//` and `///` in Dart?** `//` is an ordinary comment ignored by the doc generator. `///` is a *doc comment* — dartdoc associates it with the following declaration, parses it as Markdown, and includes it in the generated API site. Use `///` for anything documenting a public API.

🟢 **2. What is an ADR and how does it differ from an RFC?** An ADR (Architecture Decision Record) captures *one* decision in a fixed context→decision→consequences format; it's short and immutable. An RFC is a broader proposal for a design, written *before* implementation and circulated for comments. RFC = "should we build it this way?"; ADR = "we decided X, here's why, forever."

🟢 **3. Why write docs at all if code is self-documenting?** Code documents *what* it does, never *why* it does it that way, what alternatives were rejected, or what constraints shaped it. That "why" is exactly the knowledge that's expensive to lose and impossible to recover from the code alone.

🟡 **4. When would you write a design doc / RFC?** When a change is (a) hard to reverse, (b) affects multiple people or teams, (c) has non-obvious tradeoffs, or (d) will be re-questioned later. Skip it for small, local, easily-reverted changes. Rule of thumb: if getting it wrong costs more than a day, write the doc first.

🟡 **5. Why include a "Non-Goals" section?** It bounds scope explicitly, prevents scope creep, and stops reviewers from arguing about things you never intended to build. It's often more clarifying than the Goals section.

🟡 **6. What makes a doc go stale, and how do you prevent it?** Staleness happens when code changes but the doc doesn't. Prevent it with docs-as-code (docs in the repo, updated in the same PR, caught by the reviewer) and by assigning each doc an owner. Docs in a separate wiki drift fastest.

🟡 **7. What does dartdoc do with your comments?** It parses `///` comments as Markdown, uses the first sentence as the summary, resolves `[SymbolName]` references into hyperlinks, and generates a static HTML API site. Broken symbol refs produce warnings.

🔴 **8. How do you decide between a sequence diagram and a component diagram?** A sequence diagram shows *ordering over time* — who calls whom, in what order (best for "how does this flow work?"). A component diagram shows *static structure* — what depends on what (best for "what are the pieces?"). Choose by the question the reader is asking.

🔴 **9. Explain the C4 model and when to use each level.** C4 is four zoom levels: Context (system + external actors), Container (deployable apps/services/DBs), Component (parts inside one container), Code (classes). You draw the level that answers the reader's question — Context for stakeholders, Container/Component for engineers. Code-level is rarely worth drawing.

🔴 **10. Why is the *writing* of a design doc valuable even before anyone reviews it?** Writing forces you to make implicit reasoning explicit. You cannot write a clear "alternatives considered" section for alternatives you never actually considered. The act surfaces gaps in your own thinking — the doc is a thinking tool first, a communication tool second.

🔴 **11. An accepted ADR's decision turns out wrong. Do you edit it?** No. ADRs are immutable. You write a *new* ADR that supersedes it, referencing the old one and explaining what changed. The historical record of *why you once thought X* is itself valuable — editing it destroys that memory.

🟡 **12. How do you write for the right audience?** State the audience at the top and calibrate depth to it: a PM needs goals and tradeoffs, not class names; an implementer needs the interface contracts and edge cases. One doc trying to serve everyone serves no one — split it if the

audiences diverge.

Senior Engineer Tips

- **Write the RFC before you're sure.** If you already know the answer, you've skipped the thinking the doc exists to force.
- **Put the summary first.** Busy reviewers read the first paragraph and the headings. Bury the lede and you lose them.
- **A comment that says *what* is a code smell** — it usually means the code isn't clear. Fix the code; save comments for *why*.
- **Delete docs you can't keep alive.** A wrong doc is a liability. Honesty about what you'll maintain beats aspirational completeness.
- **Link liberally.** A doc that cross-references related docs (and code) becomes a navigable map, not an island.
- **Treat doc review like code review** — see Code Quality & Review. Comments on a design are cheaper than comments on a PR.

Architect Perspective

At scale, an architect cannot be in every room or on every PR. Their leverage comes almost entirely through **written artifacts**: RFCs, ADRs, and the standards docs that shape how dozens of engineers make decisions without asking. Documentation is the architect's primary instrument of *influence at a distance*.

A healthy **RFC culture** is one of the strongest signals of engineering maturity. It means: significant decisions are written down and reviewed before they're built; disagreement happens on the doc (cheap) instead of in production (expensive); and the org accumulates a searchable memory of *why* it is the way it is. The architect's job is often less "make the decision" and more "establish the process by which good decisions get made and recorded."

This connects directly to system-design practice — see System Design — where the design doc *is* the deliverable, and to the team practices in Agile & Collaboration, where RFC review is how distributed teams align asynchronously.

Summary

Documentation externalizes knowledge that otherwise dies in individual heads, raising the bus factor, cutting onboarding cost, and ending re-litigated decisions. It's a spectrum — from `///` comments to READMEs to RFCs to immutable ADRs — each matched to an audience and lifespan. Design docs and RFCs apply this *before* code exists, so thinking is reviewed while

mistakes are still cheap; the act of writing itself forces clarity. Keep docs alive with docs-as-code and clear ownership, write *why* not *what*, and reach for a diagram when relationships dominate. dartdoc turns your `///` comments into the same API site you rely on for Flutter itself.

Revision Notes

- `///` = doc comment (dartdoc reads it); `//` = ignored. First sentence must stand alone.
- Spectrum: comment → doc comment → README → API docs → runbook → wiki → changelog → RFC → ADR.
- RFC = proposal before building, socialized for comments. ADR = one decision, immutable, context→decision→consequences.
- RFC sections: Context/Problem, Goals, **Non-Goals**, Proposed Design, Alternatives, Tradeoffs, Risks, Rollout.
- Write a design doc when the change is hard to reverse, cross-team, non-obvious, or will be re-questioned.
- C4: Context → Container → Component → Code. Sequence = time; Component = structure.
- Docs-as-code prevents staleness. Say why, not what. Assign an owner.
- dartdoc: Markdown, `[symbol]` links, first-sentence summary, static HTML.

Practice Questions

1. List the documentation spectrum from smallest to largest scope, and name each one's primary audience.
2. Give a real change you'd write an RFC for, and one you deliberately wouldn't. Justify both.
3. Rewrite a "what" comment (`// loop over users`) into a useful "why" comment.
4. For "how does offline sync recover after reconnect?", pick sequence vs component diagram and explain why.
5. Why must ADRs be immutable? What do you do when the decision becomes wrong?
6. Name three team-level metrics that good documentation improves, and how you'd measure one.

Coding Questions

1. **Doc-commented API.** Write a `RateLimiter` class in Dart with `///` doc comments on the class and every public member. Use at least two `[symbol]` references and one fenced code example. Ensure it would pass the `public_member_api_docs` lint.
2. **Write an RFC.** For the feature "push-notification preferences synced across a user's devices," write a complete RFC with all standard sections (Context, Goals, Non-Goals, Proposed Design, Alternatives, Tradeoffs, Risks, Rollout).

3. **Write an ADR.** Capture the decision "store notification preferences server-side as the source of truth (not per-device)" in context→decision→consequences format, listing at least two positive and two negative consequences.
4. **Fix stale docs.** Given a function whose doc comment describes behavior the signature no longer matches, correct the comment and explain what process would have prevented the drift.

Mini Project

Write a complete design doc / RFC for an offline-sync feature in a Flutter app.

Deliverable: a single Markdown file, `docs/rfc/0001-offline-sync.md`, containing every standard section, that a real team could review.

Required sections and what to cover:

- **Status / Author / Date / Reviewers** header.
- **Context / Problem** — describe a real scenario: users edit records offline; today edits are lost or conflict. Quantify the pain if you can.
- **Goals** — e.g. edits made offline eventually reach the server; users see per-record sync state; no data loss on app kill.
- **Non-Goals** — e.g. real-time collaboration, cross-user merge, offline media upload.
- **Proposed Design** — the sync architecture: local durable store (Drift/Isar), an outbox queue, a background sync worker with backoff, conflict strategy (state your choice — e.g. last-write-wins with a `updatedAt` vector), and how the UI observes sync state via streams. Include a **C4 container diagram** in Mermaid.
- **Alternatives Considered** — at least two (e.g. full CRDT sync; server-authoritative with no offline writes) with why each was rejected.
- **Tradeoffs** — what you gain and what you knowingly give up.
- **Risks** — e.g. queue never drains, duplicate submissions, clock skew breaking last-write-wins — each with a mitigation.
- **Rollout Plan** — feature flag, staged percentage rollout, the metric you watch to decide whether to proceed or roll back.

Then, alongside it, write **one ADR** (`docs/adr/0001-conflict-strategy.md`) capturing just the conflict-resolution decision from the RFC, in context→decision→consequences format.

Success criteria: a peer who has never seen the feature can read the RFC and correctly explain *why* it's built this way, what it explicitly won't do, and what could go wrong — without asking you a single question. That is documentation doing its job.

Delivery & Release Engineering

> Delivery is the *practice* of keeping software releasable at all times through small, automated, gated batches; **release** is the *decision* to expose new behaviour to users — and feature flags exist to keep those two things separate.

Introduction

Most engineers conflate three words that mean very different things: **integrate**, **deploy**, and **release**. You *integrate* code many times a day. You *deploy* a build to an environment. You *release* a capability to users. On a server you can blur these because you control the runtime and can roll back in seconds. On mobile you cannot — once a user installs version `2.3.0`, that binary lives on their device until *they* choose to update, and the store's rollout controls are coarse and slow.

This chapter is about the *practice and decision layer* of shipping software, applied to Flutter and mobile. It is deliberately vendor-neutral: the pipeline YAML lives in module 50, the store submission steps live in module 51. Here we answer the harder questions — *which* release strategy, *why*, and *when* — and the central mobile insight that shapes all of them: **you can't easily roll back an installed app, so the levers that matter are staged rollout, remote config, feature flags, and forced-update paths.**

- Mechanics of the pipeline: ../50%20CI%20CD/README.md
- Store submission steps: ../51%20Deployment/store_setup_and_submission.md
- Deployment overview: ../51%20Deployment/README.md

Why this concept exists

Software used to ship on CDs once a year. A bug meant a recall. To reduce that risk, teams batched *everything* into a big, rare, terrifying release — which, paradoxically, made each release *more* dangerous because so much changed at once. The insight of Continuous Delivery is inverted: **the smaller and more frequent the change, the lower the risk of each one**, because less changed, the blast radius is smaller, and the fault is easy to locate.

But "release often" collides with "features take weeks to finish" and "the mobile store takes days to review and can't roll back". The resolution is **decoupling deploy from release**:

- **Deploy** the code (even unfinished code) behind an *off* flag, continuously.
- **Release** the capability later by flipping the flag — independent of any build.

This is why feature flags exist. They convert a scary, irreversible, all-at-once *release event* into a reversible, gradual, runtime *configuration change*. On mobile, where the binary is frozen on the device, this decoupling is not a nicety — it is the only practical way to control and unwind a

release without shipping a new build.

Real-world analogy

Think of a **theatre with a stage curtain**.

- **Building the set** (deploy) happens behind a closed curtain, repeatedly, whenever the crew is ready. The audience sees nothing.
- **Raising the curtain** (release) is a separate decision, made when the show is ready and the house is full.
- A **dimmer switch** (staged rollout / rollout %) lets you raise the curtain on one section of seats first.
- A **fire alarm / stop button** (kill switch) drops the curtain instantly if something catches fire — *without* tearing down the set.

Building the set and raising the curtain are different acts, controlled by different people at different times. A team that only has "open the curtain fully, all at once, and the only undo is to demolish the theatre" is doing a big-bang release. Feature flags give you the dimmer and the fire alarm.

Problem Statement

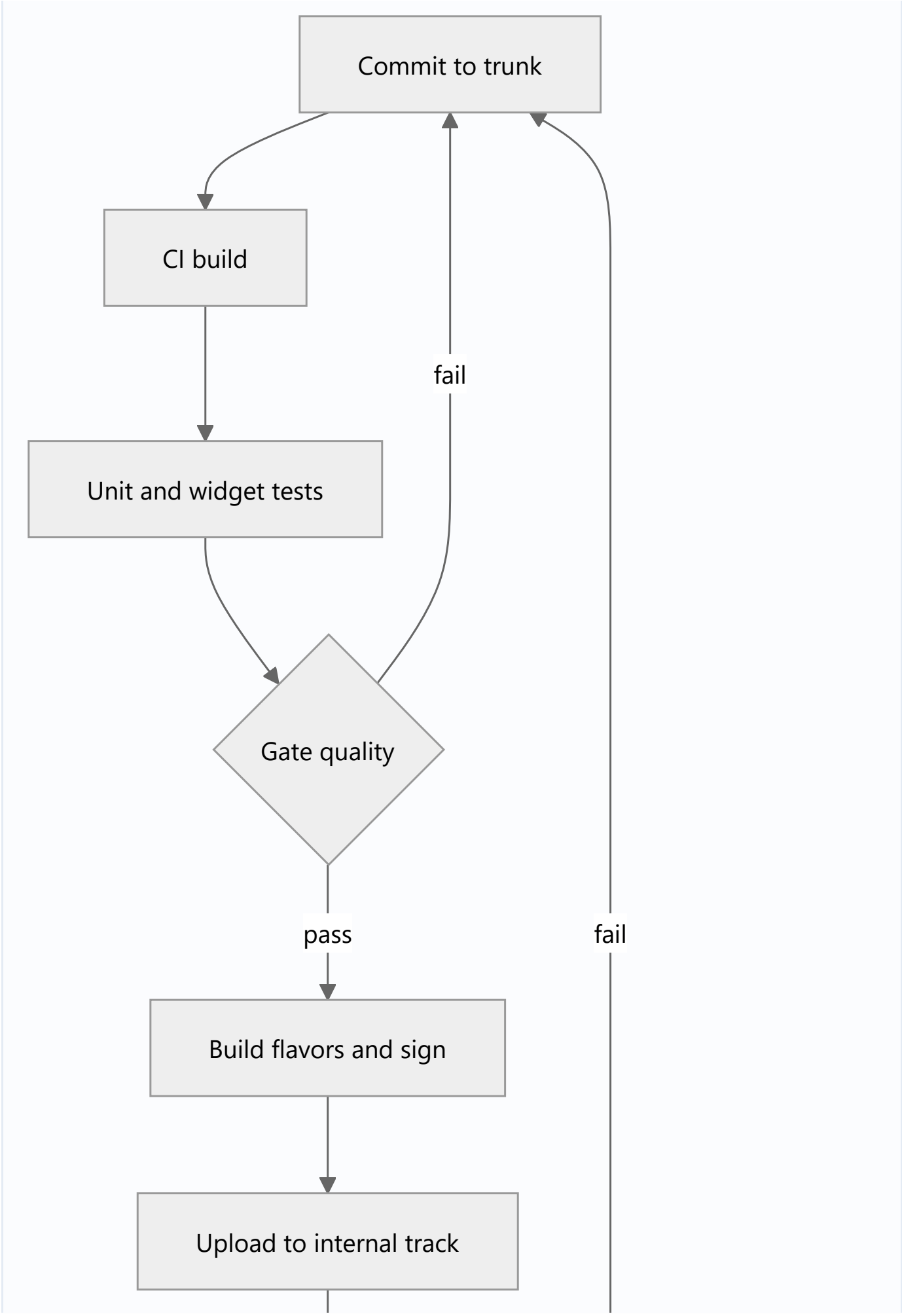
You are shipping a payments rewrite in a Flutter app with 2M installs.

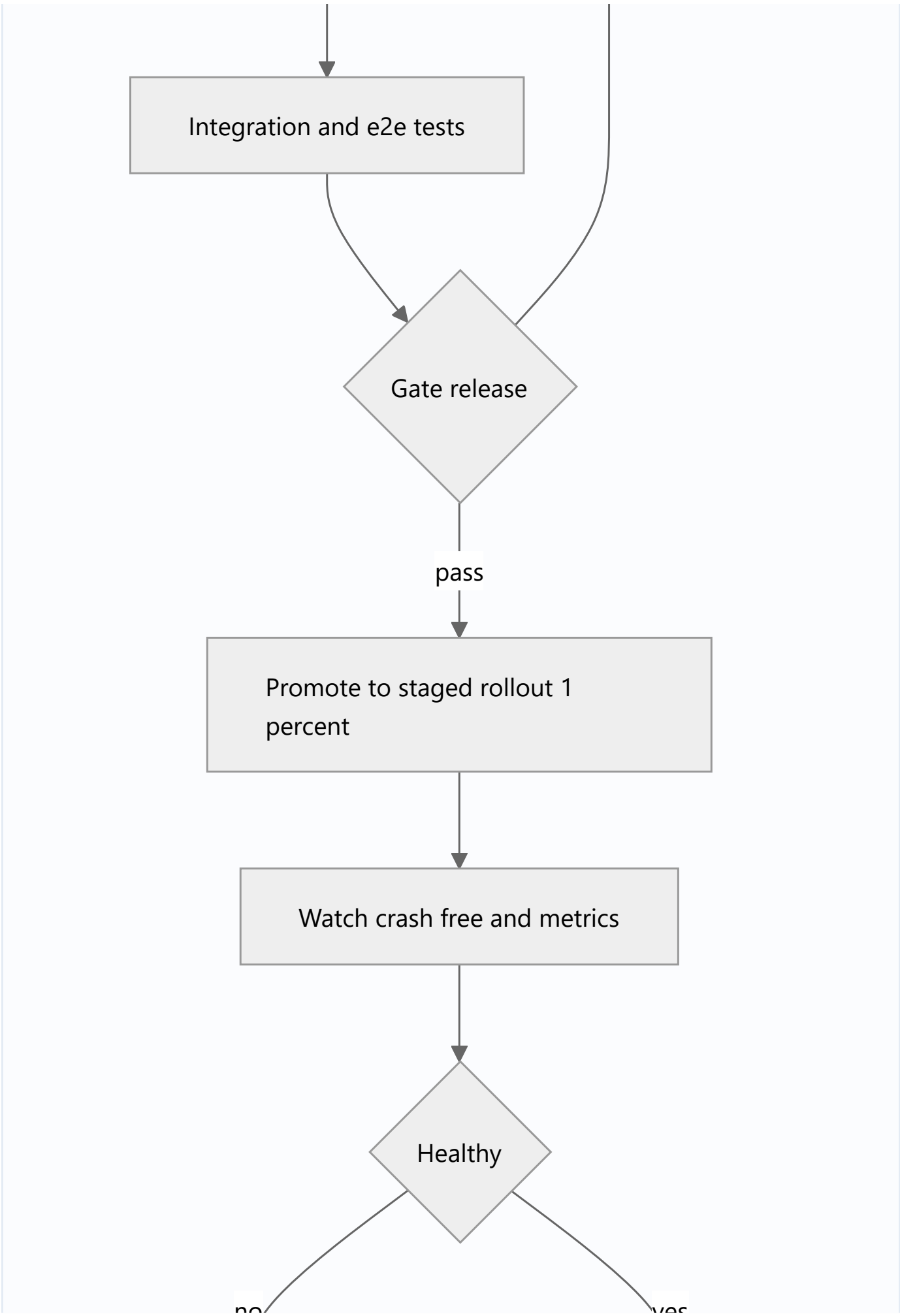
- The rewrite took 6 weeks; merging it as one giant PR is a review and integration nightmare.
- If it's broken, an App Store rollback means **submitting a new build and waiting for review** — hours to days — while every affected user is stuck on the broken binary.
- You want to expose it to 1% of users first, watch crash-free rate, then ramp.
- If crash-free rate drops, you need to disable it in **seconds, without a new build**.
- Some enterprise customers must *never* get it until legal signs off.

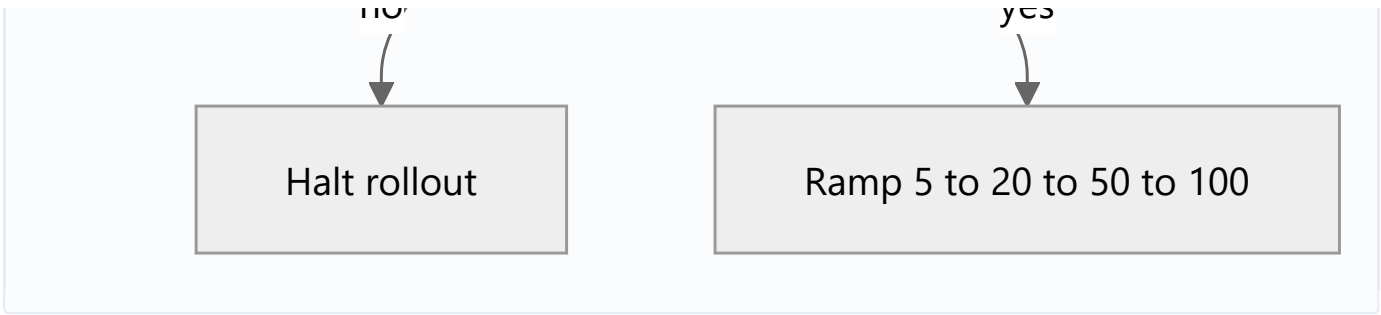
A big-bang "merge and release to 100%" approach fails every one of these constraints. The requirement set forces you toward: trunk-based development with the feature behind a flag, a deployment pipeline with gates, a staged store rollout, remote-config-backed flags for the runtime kill switch, and a forced-update path for the day you truly must retire a bad binary.

Internal Working

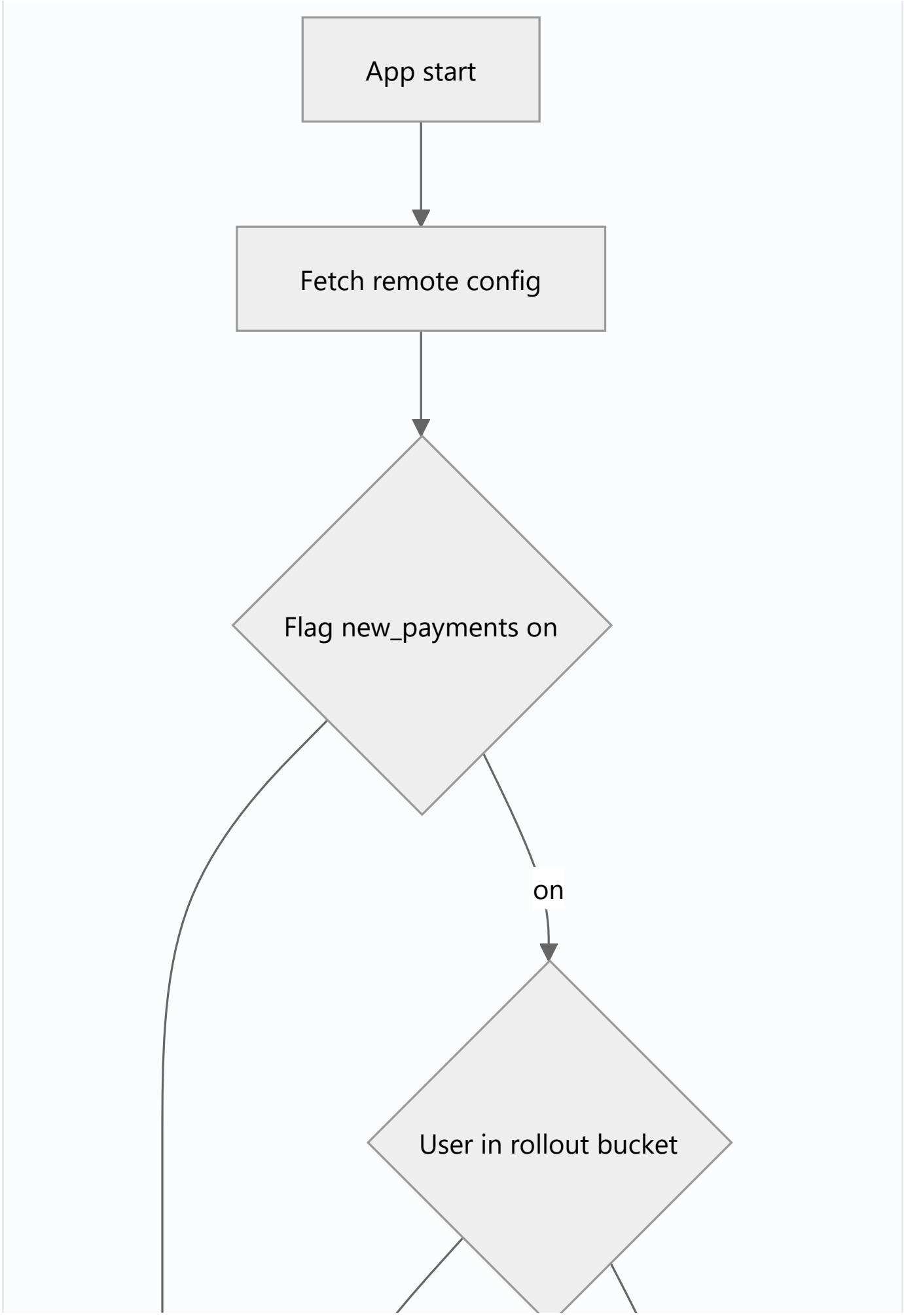
Two mechanisms combine: the **deployment pipeline** (how a commit becomes a store build, with gates) and the **feature flag** (how a runtime decision gates a code path). The pipeline gets *code onto devices*; the flag decides *whether users see it*.

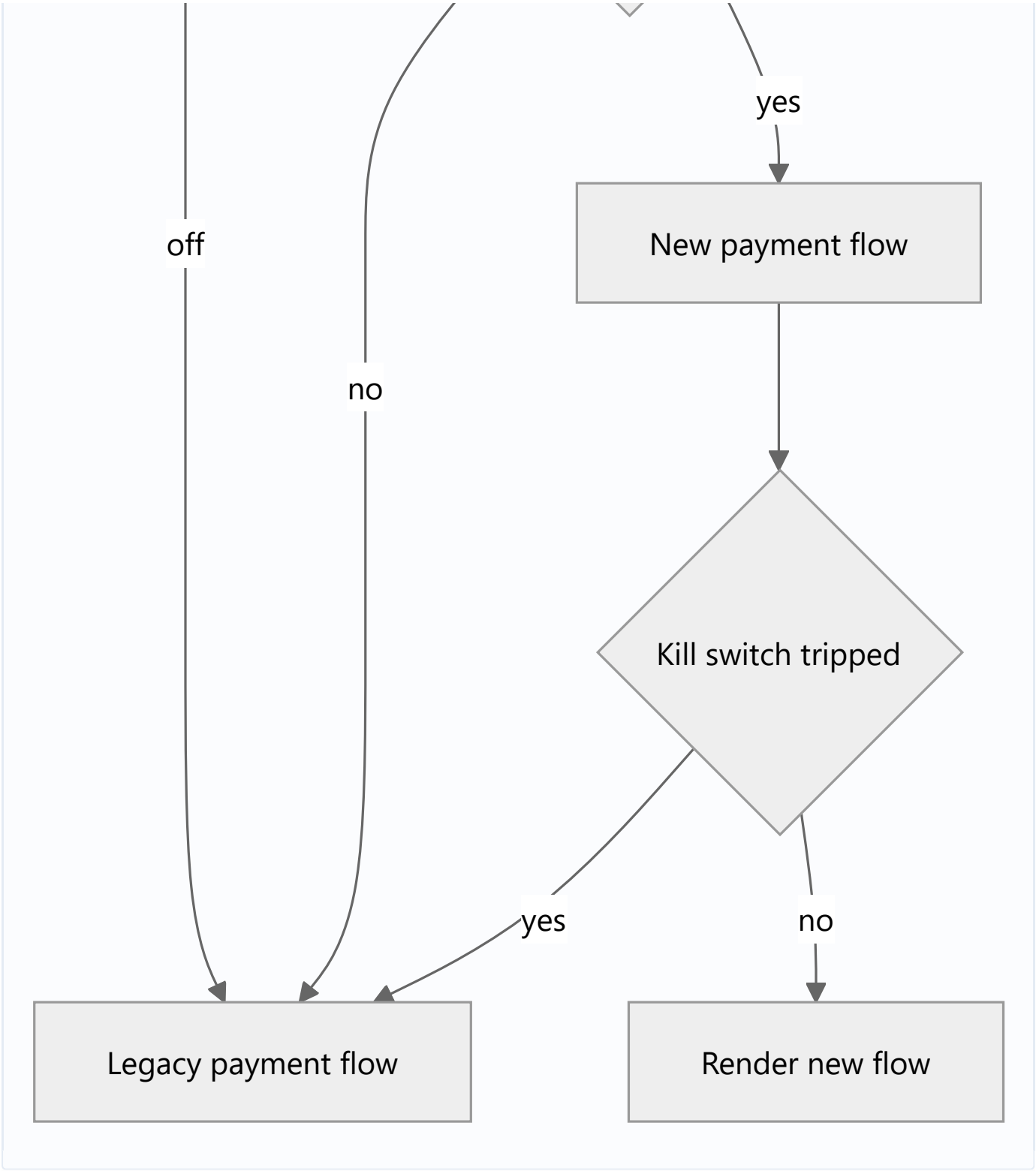

Commit to trunk
CI build
Unit and widget tests
Gate quality
fail
pass
fail
Build flavors and sign
Upload to internal track


Integration and e2e tests
Gate release
pass
Promote to staged rollout 1 percent
Watch crash free and metrics
Healthy



The flag then sits *inside* the shipped binary and decides the path at runtime:


App start
Fetch remote config
Flag new_payments on
on
User in rollout bucket



The gates are the heart of the pipeline: a change only advances when the prior stage passes. The flag is the heart of the release: the *same binary* can serve the old or new path depending on config fetched at runtime.

Memory Representation

Repurposed: here "memory" means **where release state physically lives**, because release state is data, not just code.

State	Lives in	Latency to change	Who owns
Flag defaults	Compiled into the binary (`--dart-define` / const)	New build only	Engineering
Flag overrides	Remote config service (Firebase Remote Config, LaunchDarkly, etc.)	Seconds to minutes	Eng / PM
Rollout %	App/Play store console	Minutes to hours	Release manager
Rollout bucket	Derived on-device from a stable user/install hash	Instant (deterministic)	Client logic
Min supported version	Remote config or a backend endpoint	Seconds	Eng

The key repurposing: **a release is a distributed state machine whose state is split across the binary, the remote config, and the store**. The reason flags feel powerful is that they *move* release state from the slow-to-change binary tier into the fast-to-change remote tier. The reason mobile is hard is that some state (the binary itself) can only change at store speed. See ../18%20Firebase/observability_and_messaging.md for the remote-config transport.

Compiler Behavior

Partly applicable. Release engineering touches the Dart/Flutter build in a few concrete ways:

- **Build modes**: `flutter run` (debug, JIT), `flutter build --profile`, `flutter build --release` (AOT). Store builds are always `--release`.
- **Flavors**: `--flavor prod` / `--flavor staging` select platform-side build configs (bundle id, signing, icons) so staging and prod are separate installable apps.
- **Compile-time config**: `--dart-define=FLAG_X=true` injects values read via `String.fromEnvironment` / `bool.fromEnvironment`. These are *compile-time constants*, so the tree-shaker can **eliminate the dead branch entirely** when a flavor hardcodes a flag off — a compile-time flag differs from a runtime flag precisely in that it can be shaken out.
- **Tree-shaking per flavor**: because dead `const`-guarded branches are removed, a compile-time-disabled feature adds no code to that flavor's binary. Runtime flags (remote config) cannot be shaken out — both branches ship.

The decision *compile-time vs runtime flag* is therefore also a *binary-size and security* decision, not just a convenience one.

Runtime Behavior

This is where flags actually earn their keep. At runtime the app:

1. **Fetches remote config** on start (and optionally on resume), with a sensible fetch interval and a cached fallback so a network failure degrades to last-known-good values, never to a crash.

2. **Evaluates flags** against fetched config plus a locally-derived rollout bucket (a stable hash of the install id modulo 100), so a "20% rollout" is deterministic per install and doesn't flip on every launch.
3. **Honours kill switches** — a boolean that, when true, forces the legacy path regardless of the feature flag, giving an instant runtime disable without a new build.
4. **Checks minimum supported version** — compares the running app version against a remote `min_version`; if below, shows a blocking "please update" screen (forced update).

The critical runtime property: **all of this is data-driven**, so the *same installed binary* changes behaviour when config changes — no redeploy, no review, seconds of latency.

Flutter Engine Behavior (if applicable)

Not applicable in a per-request sense — the Flutter engine does not participate in flag evaluation or rollout. The only release-relevant note: each *release build* embeds an AOT-compiled snapshot that the engine loads; there is one such snapshot per binary, so you cannot hot-swap engine-level code via a flag. Anything requiring new *native/engine* code genuinely needs a new store build — which is exactly why kill switches must live in Dart-level runtime logic, not in engine features.

Dart VM Behavior (if applicable)

Brief and relevant: **debug builds run on the Dart VM's JIT; release (store) builds are AOT-compiled** (`dart2native`-style ahead-of-time machine code, no JIT). This means:

- You cannot rely on JIT-only behaviour (hot reload, `dart:mirrors`) in a released app.
- A flag branch that only executes in release must be tested in a `--profile` / `--release` build, because JIT vs AOT can differ in timing and (rarely) in behaviour around deferred loading.
- The AOT snapshot is immutable per release — reinforcing that runtime flags gate *which compiled path runs*, they never introduce new compiled code.

Examples

A minimal, null-safe, lint-clean feature-flag abstraction with a remote-config-backed implementation, plus a forced-update check and a `--dart-define` snippet.

```
/// Contract for evaluating feature flags at runtime.
abstract interface class FeatureFlags {
  bool isEnabled(String key, {bool fallback = false});

  /// A kill switch always wins over a normal flag.
  bool isKilled(String key);
```

```
}

/// Deterministic 0..99 bucket from a stable install id.
int rolloutBucket(String installId) {
  return installId.hashCode.abs() % 100;
}

/// Remote-config-backed implementation. `config` is the raw fetched map
/// (already cached with last-known-good fallback by the transport layer).
class RemoteConfigFlags implements FeatureFlags {
  RemoteConfigFlags({
    required Map<String, Object?> config,
    required String installId,
  })  : _config = config,
        _bucket = rolloutBucket(installId);

  final Map<String, Object?> _config;
  final int _bucket;

  @override
  bool isEnabled(String key, {bool fallback = false}) {
    if (isKilled(key)) return false;
    final enabled = _config['${key}_enabled'] as bool? ?? fallback;
    if (!enabled) return false;
    final rollout = (_config['${key}_rollout_pct'] as num?)?.toInt() ?? 0;
    return _bucket < rollout;
  }

  @override
  bool isKilled(String key) => _config['${key}_kill'] as bool? ?? false;
}
```

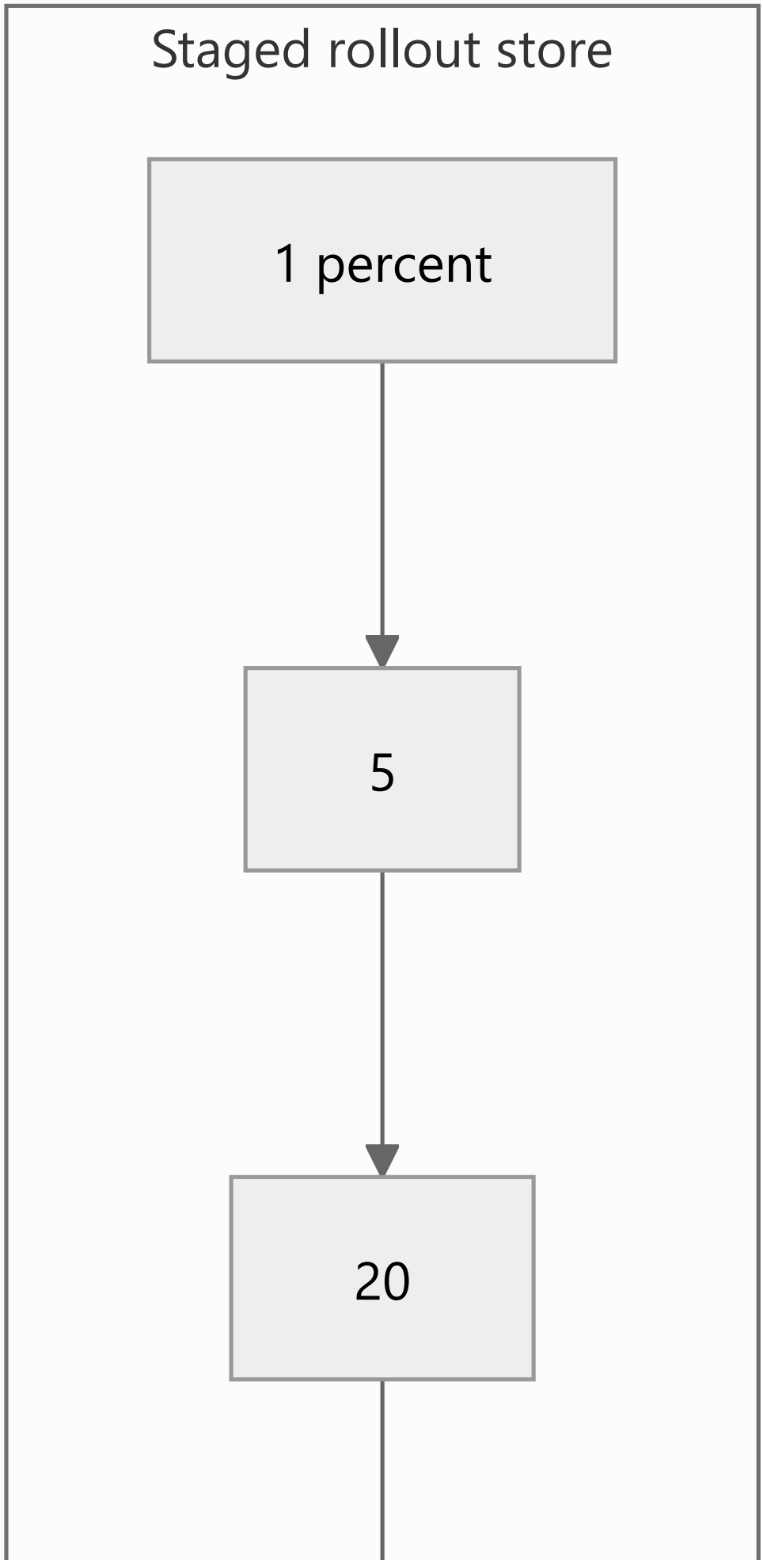
```
/// Forced-update gate. Compares running version against a remote minimum.
class VersionGate {
  const VersionGate(this.minSupported);

  /// Semantic version the backend/remote-config declares as the floor.
  final Version minSupported;

  bool mustUpdate(Version current) => current < minSupported;
```

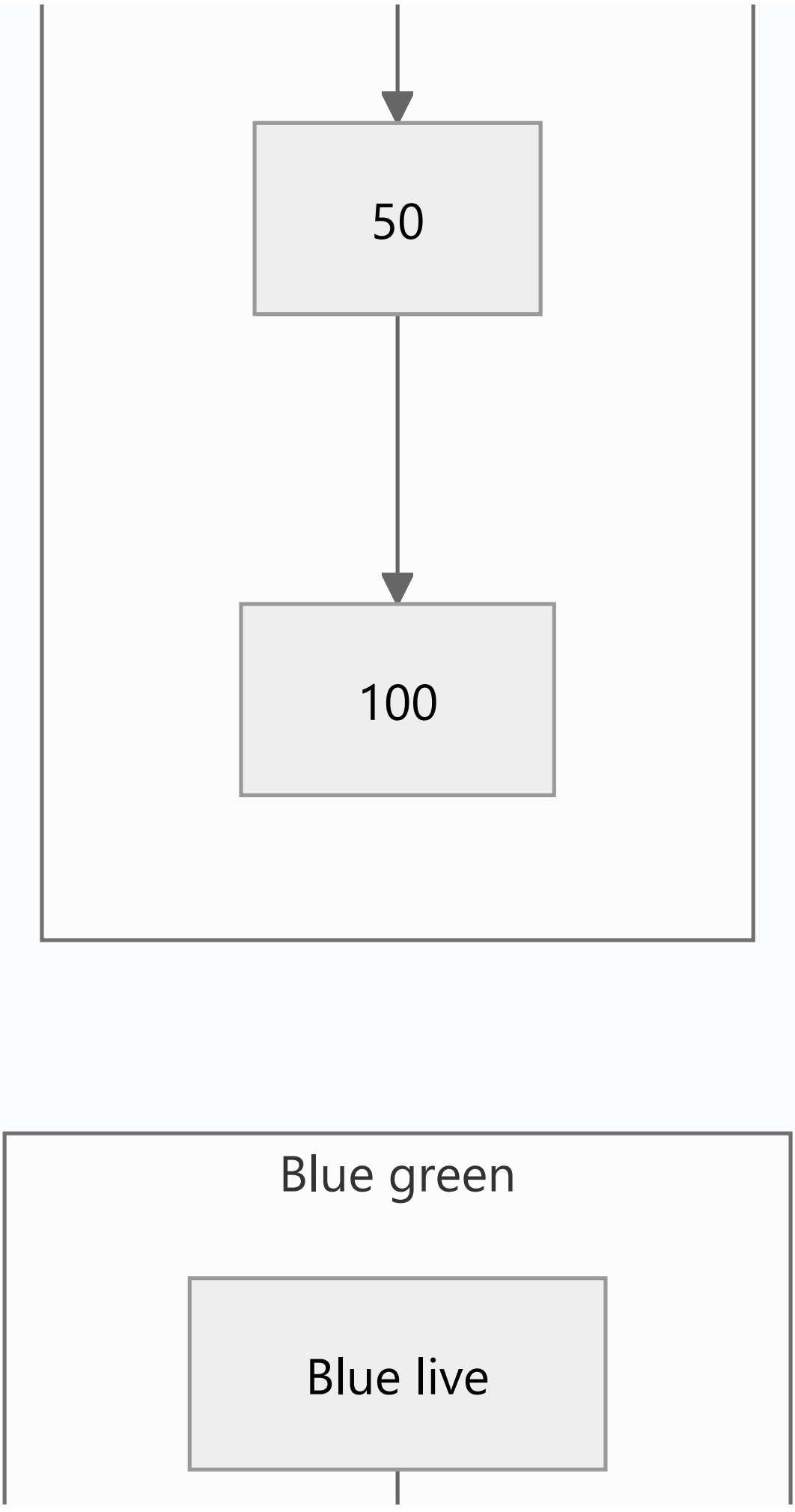
```
}

/// Tiny SemVer value type (major.minor.patch), comparable.
class Version implements Comparable<Version> {
  const Version(this.major, this.minor, this.patch);

  factory Version.parse(String raw) {
    final parts = raw.split('.').map(int.parse).toList(growable: false);
    return Version(parts[0], parts[1], parts[2]);
  }

  final int major;
  final int minor;
  final int patch;

  @override
  int compareTo(Version other) {
    final byMajor = major.compareTo(other.major);
    if (byMajor != 0) return byMajor;
    final byMinor = minor.compareTo(other.minor);
    if (byMinor != 0) return byMinor;
    return patch.compareTo(other.patch);
  }

  bool operator <(Version other) => compareTo(other) < 0;

  @override
  bool operator ==(Object other) =>
      other is Version &&
      major == other.major &&
      minor == other.minor &&
      patch == other.patch;

  @override
  int get hashCode => Object.hash(major, minor, patch);
}
```

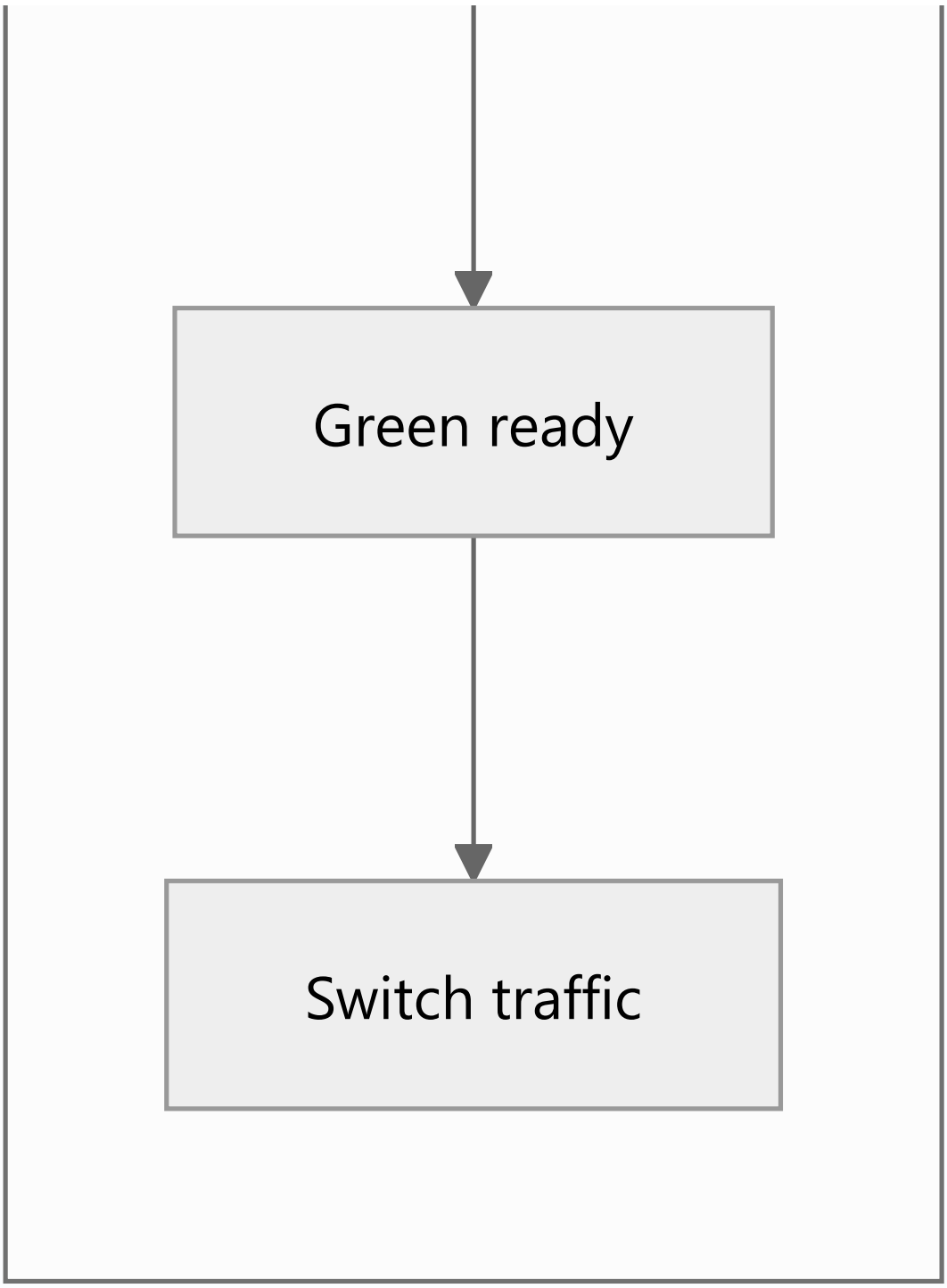
```
# Build flavor with a compile-time flag (tree-shaken when false).
flutter build appbundle \
  --release \
  --flavor prod \
  --dart-define=ENABLE_NEW_PAYMENTS=false \
  --build-name=2.3.0 \
  --build-number=430
```

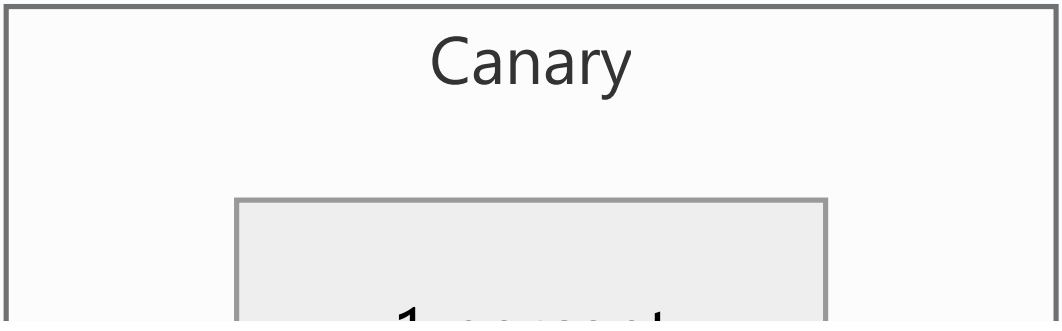
Diagrams

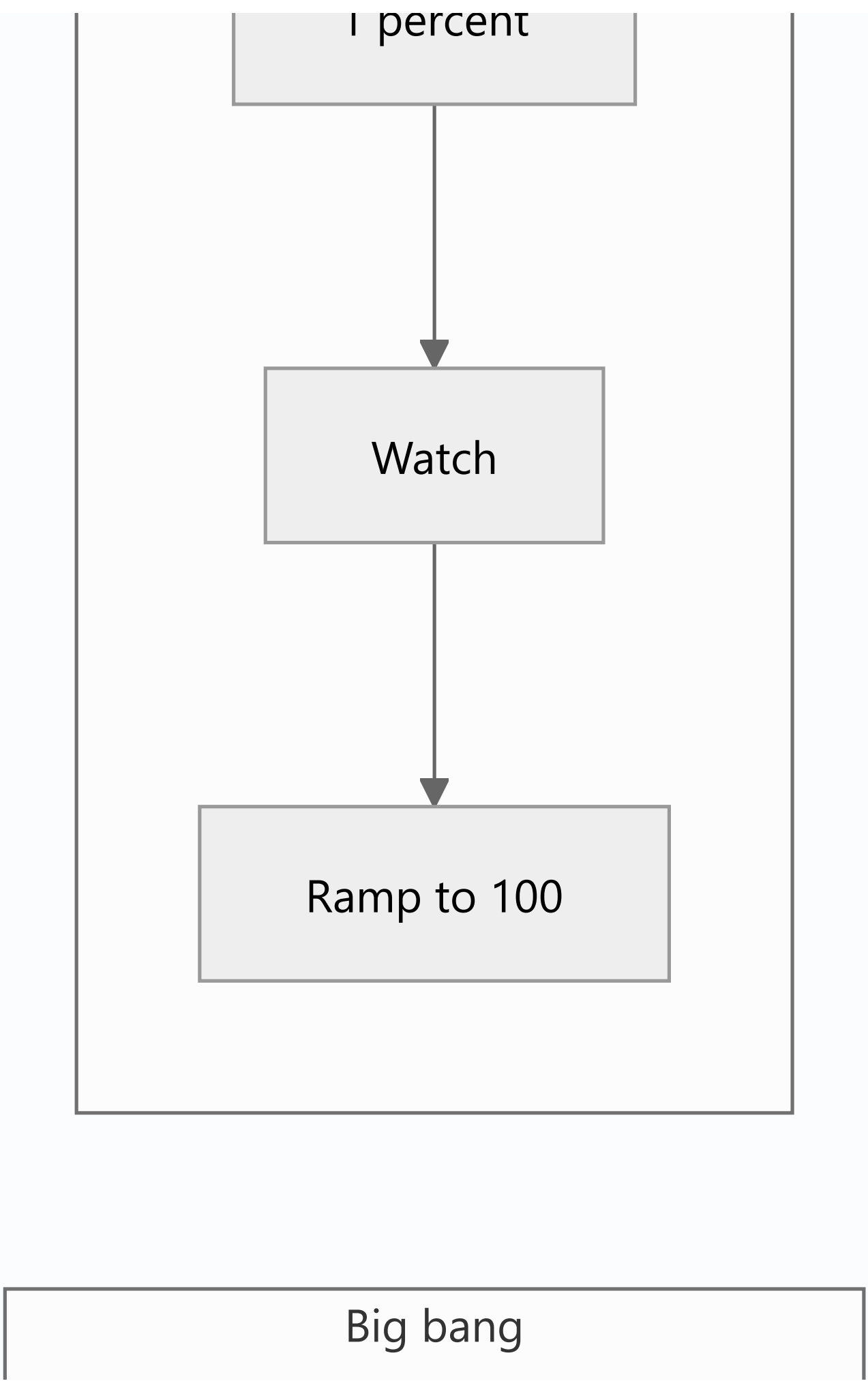
Comparison of the strategies you actually choose between on mobile:

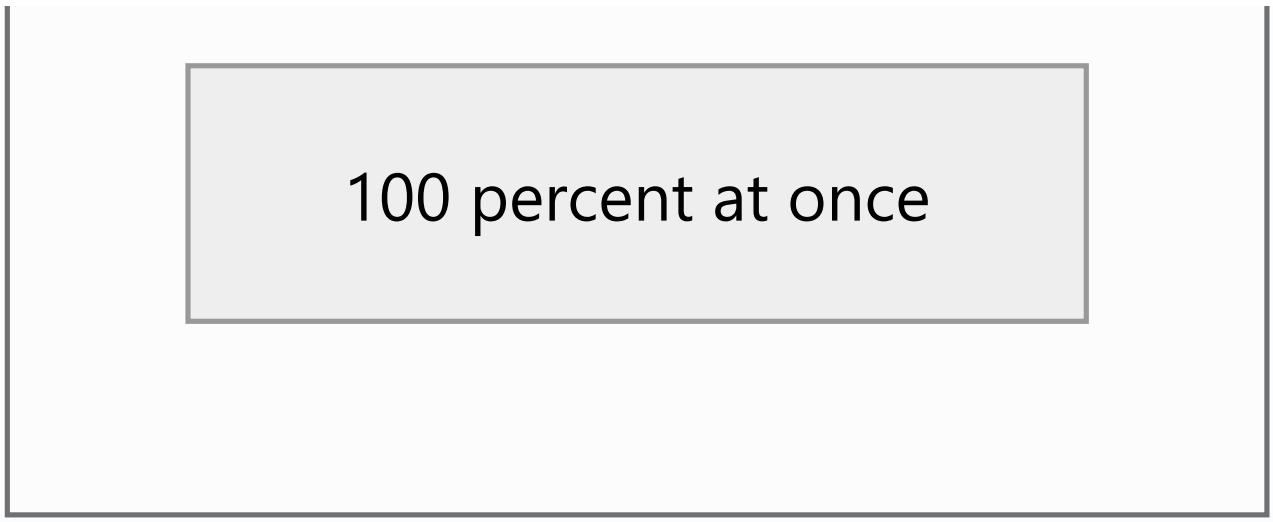

Staged rollout store
1 percent
5
20


50
100
Blue green
Blue live


Green ready
Switch traffic


Canary


1 percent
Watch
Ramp to 100
Big bang



Blue-green is a *server* pattern (two identical environments, flip a router); on mobile the closest analogue is the flag flip — old path is "blue", new path is "green", both live in one binary. **Staged rollout is the store-native form of canary** and is the strategy you use for the binary itself, while feature flags give you canary-*within*-a-binary.

Common Mistakes

Mistake	Why it hurts	Fix
Treating deploy == release	You can't ship unfinished code; releases become big and rare	Deploy behind an off flag; release by flipping it
No kill switch	A bad release needs a new build + store review to disable	Every risky feature ships with a remote kill switch
Never cleaning up flags	Flag debt: combinatorial code paths, dead branches, confusion	Track flag age; remove after full rollout; treat as a ticket
100% rollout on day 1	Full blast radius; no early signal	Staged rollout 1→5→20→50→100 with health checks
No forced-update path	You can never retire a truly broken binary	Ship a min-version gate from v1, before you need it
Compile-time flag for something you must toggle live	Can't change without a new build	Use runtime/remote flag for anything you may disable in prod
Flag with no default / no fallback	Network failure at start = undefined behaviour	Cached last-known-good + safe compile-time default

Best Practices

- **Trunk-based development**: short-lived branches, merge to trunk daily, keep trunk releasable — this is what *enables* continuous delivery.
- **Small batches**: many small releases beat one big one; smaller blast radius, easier bisection.

- **Everything automated & gated**: no manual step in the path to production except the deliberate *release* decision.
- **Flag hygiene**: name flags with owner + created date, set an expiry, and delete them once at 100% for two stable releases.
- **Kill switch first**: add the off-switch before the feature, not after the incident.
- **Ship the forced-update gate early**, ideally in your very first release, since you can only *use* it in versions that already contain it.
- **Deterministic buckets**: hash a stable install id so rollout % is stable per user.
- **Observe every release**: wire crash-free rate and key metrics as rollout gates — see ./observability_as_practice.md.

Performance

Repurposed: the performance of a *delivery system* is measured by the **DORA metrics**, not by latency.

Metric	What it measures	Elite ballpark
Deployment frequency	How often you ship to prod	On demand / multiple per day
Lead time for changes	Commit → running in prod	Less than one day
Change failure rate	% of releases causing a failure	0–15%
MTTR (time to restore)	How fast you recover from a bad release	Less than one hour

The mobile twist: **MTTR is bounded by store review**, so your *real* MTTR lever is the runtime kill switch / remote config, which restores in seconds — not a new submission. A team optimising mobile delivery invests in flags precisely to decouple MTTR from store latency.

Advantages

- Releasable at any time; the *release* becomes a business decision, not an engineering scramble.
- Small blast radius; faults are cheap to locate and cheap to unwind.
- Fast recovery via flags/remote config, independent of store speed.
- Progressive exposure (canary/staged) surfaces problems on 1% before 100%.
- Experimentation (A/B) and permissioning (enterprise gating) fall out of the same flag machinery.

Disadvantages

- **Flag debt** is real: unbounded flags create combinatorial, untested code paths.
- Testing complexity: every flag combination is a potential state to verify.

- Requires discipline and infrastructure (remote config, observability, rollout tooling).
- On mobile you still can't fix *native* bugs without a new build — flags only gate Dart-level paths.
- Remote-config dependency introduces a new failure mode you must design fallbacks for.

Interview Questions

🟢 **1. What's the difference between Continuous Integration, Continuous Delivery, and Continuous Deployment?** CI = merge and auto-test small changes frequently. Continuous Delivery = every change is *automatically proven releasable* but release is a manual decision. Continuous Deployment = every change that passes automatically goes to production with no human gate.

	Continuous Integration	Continuous Delivery	Continuous Deployment
Auto-merge + test	yes	yes	yes
Always releasable	—	yes	yes
Release step	manual	manual button	automatic

🟢 **2. What does "decouple deploy from release" mean?** Deploy = get the code onto the runtime/device (possibly off). Release = expose the behaviour to users. Feature flags let you deploy code weeks before you release it, and release without deploying.

🟢 **3. Name the feature-flag types.** Release flags (temporary, hide unfinished work), ops flags (kill switches, circuit breakers), experiment flags (A/B), permission flags (entitlement/segment gating). They differ in lifespan and owner.

🟡 **4. Why do feature flags matter *more* on mobile than on a server?** Because you can't roll back an installed binary — a rollback needs a new build plus store review (hours to days). A remote-config-backed flag disables the bad path in seconds without any new build, so on mobile the flag *is* your rollback.

🔴 **5. How do you roll back a bad mobile release?** You (usually) don't "roll back" — you **halt the staged rollout** in the store so no *new* users get it, then **flip the kill switch / remote config** to force existing users onto the safe path instantly. If the bug is native and unflaggable, you ship a fixed build (optionally with an *expedited review*) and use the **forced-update** gate to push users off the broken version. Restoring the old binary to already-updated users is not directly possible; staged rollout + flags + forced update are the real levers.

🟡 **6. Big-bang vs canary vs blue-green vs staged rollout — when each?** Big-bang: trivial/low-risk changes. Canary: expose to a small % and watch. Blue-green: server-side instant switch between two environments (mobile analogue = flag flip). Staged/phased rollout: the store-native canary for the binary itself. Mobile risky features = staged rollout of the binary + canary via flags inside it.

🟡 **7. What is flag debt and how do you manage it?** Flags that outlive their purpose, leaving dead/combinatorial paths. Manage with naming conventions (owner + date), expiry dates, a "remove flag" ticket created with the flag, and deletion once fully rolled out and stable.

🔴 **8. Compile-time vs runtime flag — trade-offs?** Compile-time (`--dart-define`) is tree-shaken out (smaller binary, code not present — good for security/kill of unreleased code) but needs a new build to change. Runtime (remote config) ships both branches and can toggle live in seconds. Use compile-time for build-variant differences, runtime for anything you may need to disable in production.

🟡 **9. What are the DORA metrics and which is special on mobile?** Deployment frequency, lead time for changes, change failure rate, MTTR. On mobile MTTR is throttled by store review, so kill switches/remote config are what actually keep MTTR low.

🟢 **10. App bundle vs APK?** An Android App Bundle (AAB) is uploaded to Play, which generates optimized per-device APKs (smaller downloads). APK is a single installable package. Stores require AAB for new apps; APK is for sideloading/testing.

🔴 **11. How do staged rollout percentages actually reach specific users?** The store selects a random subset of eligible devices for the given %. It's not user-choosable and is coarse; for precise targeting you layer a feature flag with a deterministic install-id bucket *inside* the shipped build.

🟡 **12. Why ship a forced-update mechanism before you need it?** Because it can only act on versions that *already contain* it. If v1 lacks the gate, you can never force v1 users to update; the capability must be present in the version you later want to retire.

Senior Engineer Tips

- The kill switch you didn't ship is the incident you can't stop. Add it *with* the feature.
- Make rollout buckets deterministic per install, or your "20%" flickers and your metrics lie.
- A flag without an owner and an expiry is a future landmine — attach both at creation.
- Test the *off* path as rigorously as the *on* path; the fallback is what runs during an incident.
- Cache remote config; a cold start during a network blip must degrade to last-known-good, never to a crash or an undefined flag state.
- "It's deployed" is not "it's released." Say which you mean in every standup.

Architect Perspective

At org scale, release strategy is fundamentally a **risk-management decision**, and the architect's job is to make risk *adjustable* rather than binary. The core move is decoupling deploy from release across the whole org:

- Engineering ships continuously to trunk and to the store behind flags — cadence is decoupled from feature completeness.
- Product/release management owns the *release* decision and the rollout dial — business risk is decoupled from engineering cadence.
- SRE/observability owns the gates — a rollout that breaches a metric halts automatically.

This separation means a 6-week feature and a 1-line fix ship through the *same* pipeline at the *same* cadence; only the *release* moment differs. On mobile the architect additionally treats **remote config + kill switch + forced-update as first-class infrastructure**, because they are the only mechanisms that give sub-store-review MTTR. The strategic principle: **invest in the levers that make recovery fast and exposure gradual, because on mobile you cannot buy those back after the binary has shipped.**

Summary

- Integrate ≠ deploy ≠ release. Continuous Delivery keeps you *always releasable*; Continuous Deployment removes the human gate.
- Delivery is a *practice*: small batches, everything automated and gated, trunk-based development, releasable at any time.
- **Feature flags decouple deploy from release** — the central insight — turning an irreversible release event into a reversible runtime config change.
- Flag types: release, ops (kill switch), experiment, permission. Compile-time flags tree-shake out; runtime flags toggle live.
- Release strategies: big-bang, canary, blue-green, rolling, staged/phased. Mobile uses **staged rollout for the binary + flags for canary inside it**.
- **You can't easily roll back mobile**: halt the staged rollout + flip the kill switch + forced-update for truly broken binaries.
- Release state is distributed across binary, remote config, and store; DORA metrics measure the system, and mobile MTTR depends on flags, not store speed.

Revision Notes

- CI = test on merge. CD (delivery) = always releasable, manual release. CD (deployment) = auto release.
- Deploy = code onto device. Release = behaviour to users. Flags separate them.
- 4 flag types: release / ops / experiment / permission.
- Compile-time flag → tree-shaken, needs new build. Runtime flag → both branches ship, toggles in seconds.
- Mobile "rollback" = halt rollout + kill switch + forced update. Not a true rollback.

- SemVer `major.minor.patch` for the app; monotonic build number for the store.
- DORA: deployment frequency, lead time, change failure rate, MTTR.
- Ship the forced-update gate in v1 — it only works in versions that already contain it.

Practice Questions

1. Explain to a PM why "we deployed it" does not mean "users have it."
2. Draw the deployment pipeline for your app and mark every gate.
3. Classify three flags in your codebase as release/ops/experiment/permission and give each an expiry.
4. Your crash-free rate drops during a 20% rollout. Write the exact sequence of actions you take.
5. Argue when a compile-time flag is *correct* and when it is a mistake.
6. Compute your team's four DORA metrics for last month; identify the weakest and the mobile-specific reason.
7. Describe how a "20% rollout" reaches deterministic users end-to-end (store % + in-app bucket).

Coding Questions

A. Implement a feature-flag gate. Acceptance criteria:

- `FeatureFlags.isEnabled(key)` returns `false` when the kill switch for `key` is set, regardless of the enabled flag.
- Respects a `_rollout_pct` value using a deterministic per-install bucket.
- Falls back to a supplied default when the key is absent.
- Null-safe, no `dynamic`, lint-clean. (See `RemoteConfigFlags` above as a reference solution.)

B. Implement a min-version forced-update check. Acceptance criteria:

- `mustUpdate(current)` returns `true` iff `current < minSupported` under SemVer ordering.
- Parses `major.minor.patch` strings; `2.3.0 < 2.10.0` must be true (numeric, not lexical).
- Pure/deterministic, no I/O in the comparison. (See `VersionGate` / `Version` above.)

Extend B: add a `latest` version and a `bool shouldSuggestUpdate` that is soft (dismissible) when `current < latest` but `current >= minSupported`.

Mini Project

Staged-rollout release plan for a Flutter feature, with flags and a kill switch.

Design a release plan for a new `NewCheckoutFlow` feature.

Design deliverables:

1. **Flag design** — one release flag `new_checkout_enabled`, one ops kill switch `new_checkout_kill`, one rollout key `new_checkout_rollout_pct`, backed by remote config with cached fallback and a safe `false` default.
2. **Rollout schedule** — internal track → closed test → staged rollout `1 → 5 → 20 → 50 → 100`, each step gated on crash-free rate ≥ 99.5% and checkout-success metric not regressing.
3. **Kill path** — set `new_checkout_kill = true` to force all users to legacy in seconds; document that this needs no new build.
4. **Store-halt path** — how to pause the staged rollout in the console if the *binary* itself is bad.
5. **Forced-update path** — bump `min_version` when a native fix ships and the old binary must be retired.

Partial implementation (wire into the examples above):

```
class CheckoutRelease {
  const CheckoutRelease(this._flags);
  final FeatureFlags _flags;

  bool get useNewCheckout =>
      _flags.isEnabled('new_checkout', fallback: false);
}

// Router usage:
// final release = CheckoutRelease(remoteFlags);
// return release.useNewCheckout ? const NewCheckoutFlow() : const LegacyCheckout();
```

Acceptance criteria:

- Flipping `new_checkout_kill` in remote config sends 100% of users to `LegacyCheckout` on their next config fetch, with no new build.
- A fresh install with no network gets the safe default (`LegacyCheckout`) and never crashes.
- The rollout % is stable per install across launches.
- The plan documents *halt* (not "rollback") as the store-level response and *forced update* as the last resort.

Cross-references: pipeline mechanics in ../50%20CI%20CD/README.md; store tracks, signing, versioning, phased rollout in ../51%20Deployment/store_setup_and_submission.md and ../51%20Deployment/README.md; remote config transport and crash reporting in ../18%20Firebase/observability_and_messaging.md; metric-gated rollouts in ./observability_as_practice.md.

Observability as an Engineering Practice

> Observability is the discipline of designing a system so its external outputs — logs, metrics, and traces — let you answer new, unanticipated questions about its internal state without shipping new code.

Introduction

Monitoring tells you *whether* the thing you were worried about happened. Observability lets you *investigate* the thing you never thought to worry about. This distinction is the whole chapter.

Most teams start with monitoring: a dashboard of CPU, a crash-rate alert, a "requests per second" graph. That answers **known-unknowns** — questions you already knew to ask. But production fails in ways nobody predicted. Observability is the property that, when a novel failure appears, your telemetry is rich enough to slice, filter, and correlate your way to the cause — without a redeploy.

This file lives in the *Software Engineering Practices* module. It is deliberately about the **practice and discipline** layer, not the tooling. The mechanics of monitoring live in Monitoring, and the mechanics of emitting logs live in Logging. Here we care about: how you decide what to emit, how you alert on it, how you respond when it fires, and how reliability targets govern how fast you ship.

Why this concept exists

Software behaves differently in production than on your machine. The gap grows with scale, concurrency, third-party dependencies, and — for mobile — the fact that your code runs on a device you do not own, on a network you do not control, for a user you cannot interview.

Observability exists because **you cannot predict every failure mode**, so you cannot pre-build a dashboard for every one. Predefined dashboards catch the failures you imagined. The outage that pages you at 3am is, almost by definition, the one you didn't imagine. If your system only emits the numbers you thought to graph, you are blind to everything else.

The practice discipline exists on top of that because raw telemetry is worthless without judgement: log everything and you get a noisy, expensive, PII-leaking haystack; alert on everything and your team stops reading alerts; graph everything and nobody looks at any of it. Observability-as-practice is the set of decisions that keep signal high and cost, noise, and risk low.

Real-world analogy

A **car dashboard** is monitoring: the fuel gauge, speedometer, and check-engine light are predefined signals for predefined concerns. They are excellent — until something happens the manufacturer didn't put a light for.

A mechanic's diagnostic port (OBD-II) is observability: it exposes the engine's internal state as structured data so a technician can ask questions the dashboard never anticipated — "why does the misfire only happen when the engine is warm and turning left?" The car was *designed to be interrogable*. That design choice — instrumenting the engine so unforeseen questions are answerable — is exactly what observability asks of software.

Monitoring is the dashboard light. Observability is the diagnostic port.

Problem Statement

You ship a Flutter app. A subset of users report that "checkout sometimes hangs." You have:

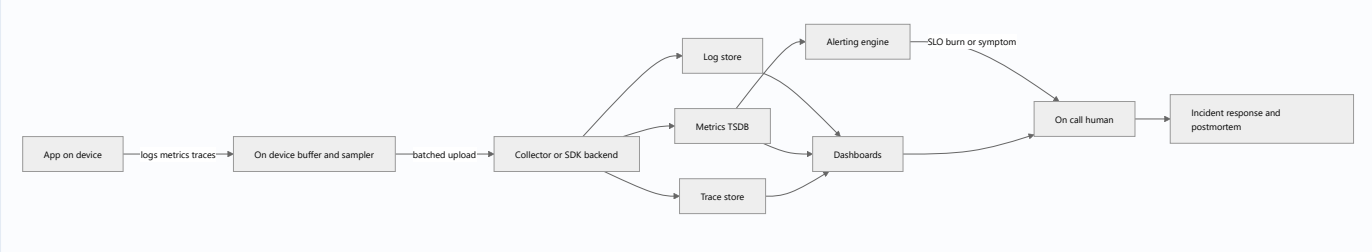
- A crash-free-users metric sitting at 99.6% (looks fine).
- A dashboard of average API latency (looks fine).
- No idea which users, which flow, which network conditions, or which backend call.

Averages hid a tail. The crash metric didn't fire because it wasn't a crash — it was an ANR-like hang. You cannot reproduce it. You cannot ask "show me the checkout traces for users on 3G in region X where the payment span exceeded 5s" because you never emitted spans, correlation IDs, or the dimensions you'd need to slice by.

The problem this chapter solves: **how to instrument, alert, and respond so that the next "sometimes it hangs" is answerable from data you already collect.**

Internal Working

Telemetry flows from the device, through a collector, into backends, and out to humans as dashboards and alerts. The three pillars converge during an incident.



Key points in the flow:

- **On-device buffer and sampler** — telemetry is not sent immediately; it is batched, sampled, and held while offline. This is the biggest difference from server observability.

- **Collector** — normalizes, enriches (adds app version, OS, region), and fans out to the right store per pillar.
- **Alerting engine** reads primarily from **metrics** (cheap to evaluate continuously), fires on **symptoms / SLO burn**, and pages a human.
- The human uses **dashboards + traces + logs together** to diagnose, then feeds learning back via a **postmortem**.

Memory Representation

*Repurposing note: "memory representation" here means the **state each telemetry type retains** — the shape of what is stored — plus what the device holds before upload. It is not about Dart heap layout.*

Pillar	State shape retained	Read pattern
Logs	Append-only **event stream** — discrete, timestamped, high-detail records. High volume, retained short.	Search / filter by field
Metrics	**Time-series** — pre-aggregated numbers keyed by name + labels, bucketed over time. Low volume, retained long.	Aggregate over time
Traces	**Span tree** — a causal parent/child tree of timed operations sharing one trace ID. Medium volume, often sampled.	Follow one request end to end

On-device state before upload:

- A **ring buffer** of recent log/breadcrumb events (bounded, oldest dropped) attached to the next crash report.
- A **counter/gauge aggregation buffer** so metrics are summed locally and flushed periodically, not sent per-event.
- A **persistence queue** on disk so telemetry survives app kill and network loss, replayed on next launch. This is why mobile telemetry arrives **delayed and out of order**.

Compiler Behavior

Not applicable — because observability is a runtime and operational concern, not a language-level one. The Dart compiler (`dart compile`, the AOT/JIT pipelines) does nothing observability-specific: it does not inject spans, aggregate metrics, or wire crash handlers. Instrumentation is ordinary application code and SDK calls that exist at runtime. The only adjacent compile-time detail is that AOT release builds strip symbols, so crash stack traces need **symbolication** via uploaded debug symbols (dSYM / ProGuard mapping / Dart symbol files) — but that is a build-artifact concern handled by your reporting SDK, not compiler semantics.

Runtime Behavior

At runtime, telemetry is emitted through **hooks** and **error surfaces**, then buffered and shipped:

- **Explicit instrumentation** — you call `log.info(...)`, `metrics.increment(...)`, `tracer.startSpan(...)` at meaningful points.
- **Global error hooks** capture the unforeseen (see Dart VM Behavior below): `FlutterError.onError`, `PlatformDispatcher.instance.onError`, and a `runZonedGuarded` boundary around `runApp`.
- **Zones** (`Zone`) let you install a `runZonedGuarded` handler that catches async errors escaping the framework, and can also override `print` to route stdout into your structured logger.
- **Batching & sampling** — the SDK accumulates events and flushes on a timer, on batch-size threshold, or on background transition. Traces are typically **head-sampled** (decide at span start) to cap volume.
- **Offline buffering** — events persist to disk and replay when connectivity returns; expect duplicates and clock skew, and design idempotent, timestamp-tolerant analysis.

The runtime contract: instrumentation must be **cheap and non-blocking** on the hot path (frame builds, gesture handlers). Do the work of formatting/uploading off the critical path.

Flutter Engine Behavior (if applicable)

Applicable. The Flutter engine is itself a telemetry source for **performance observability**:

- The engine reports **frame timings** via `SchedulerBinding.instance.addTimingsCallback`, exposing `buildDuration` and `rasterDuration` per frame. Frames exceeding the budget (~16ms at 60Hz) are **jank**.
- The engine emits **timeline events** consumable by DevTools' performance view and by `dart:developer` `Timeline` spans — the on-device analogue of distributed tracing for the UI thread and raster thread.
- Real-user jank metrics (e.g., "slow frame %" and "frozen frame %") are derived from these callbacks and are a first-class mobile SLI, not a lab-only number.

So the engine gives you the raw signal for a **UI-performance SLO**; you decide sampling and what to ship.

Dart VM Behavior (if applicable)

Applicable and central. The Dart VM's error-propagation model is *how* uncaught failures reach your reporter:

- **Synchronous framework errors** surface through `FlutterError.onError`.

- **Uncaught async errors in the platform dispatcher** (errors with no Flutter/zone handler) surface through `PlatformDispatcher.instance.onError` — returning `true` marks them handled.
- **Errors escaping a guarded zone** surface through the `runZonedGuarded` callback, which is why `runApp` is wrapped in one.
- **Isolate errors** — errors in other isolates don't cross automatically; add an `Isolate.current.addErrorListener` / `addOnErrorListener` port if you spawn isolates.

Together these three-to-four surfaces are the VM-level net that turns "the app just died" into a structured, symbolicated crash event.

Examples

Structured logging with levels + correlation ID, global error wiring to a reporter, and a simple counter — null-safe and lint-clean.

```
import 'dart:async';
import 'package:flutter/foundation.dart';
import 'package:flutter/widgets.dart';

/// Severity levels, aligned with module 39 logging conventions.
enum LogLevel { debug, info, warning, error }

/// Minimal structured logger: emits a single map per event so backends can
/// index by field. A correlation id links every log in one user flow.
class StructuredLogger {
  StructuredLogger(this._sink);
  final void Function(Map<String, Object?> event) _sink;

  void log(
    LogLevel level,
    String message, {
    required String correlationId,
    Map<String, Object?> fields = const <String, Object?>{},
  }) {
    _sink(<String, Object?>{
      'ts': DateTime.now().toUtc().toIso8601String(),
      'level': level.name,
      'msg': message,
      'correlationId': correlationId,
      ...fields,
    });
```

```
  }
}

/// A tiny counter metric that aggregates locally and reports on flush.
class Counter {
  int _value = 0;
  void increment([int by = 1]) => _value += by;
  int drain() {
    final int v = _value;
    _value = 0;
    return v;
  }
}

/// Pretend crash reporter (Crashlytics/Sentry stand-in).
Future<void> reportToCrashReporter(
  Object error,
  StackTrace stack, {
  bool fatal = false,
}) async {
  // In real code: FirebaseCrashlytics.instance.recordError(...) etc.
  debugPrint('CRASH fatal=$fatal: $error');
}

void main() {
  // 1. Synchronous Flutter framework errors.
  FlutterError.onError = (FlutterErrorDetails details) {
    FlutterError.presentError(details);
    unawaited(reportToCrashReporter(details.exception, details.stack ?? StackTrace.current, fatal: true));
  };

  // 2. Uncaught async errors at the platform dispatcher.
  PlatformDispatcher.instance.onError = (Object error, StackTrace stack) {
    unawaited(reportToCrashReporter(error, stack, fatal: true));
    return true; // handled
  };

  // 3. Everything escaping the zone around runApp.
  runZonedGuarded<void>(
    () => runApp(const _App()),
```

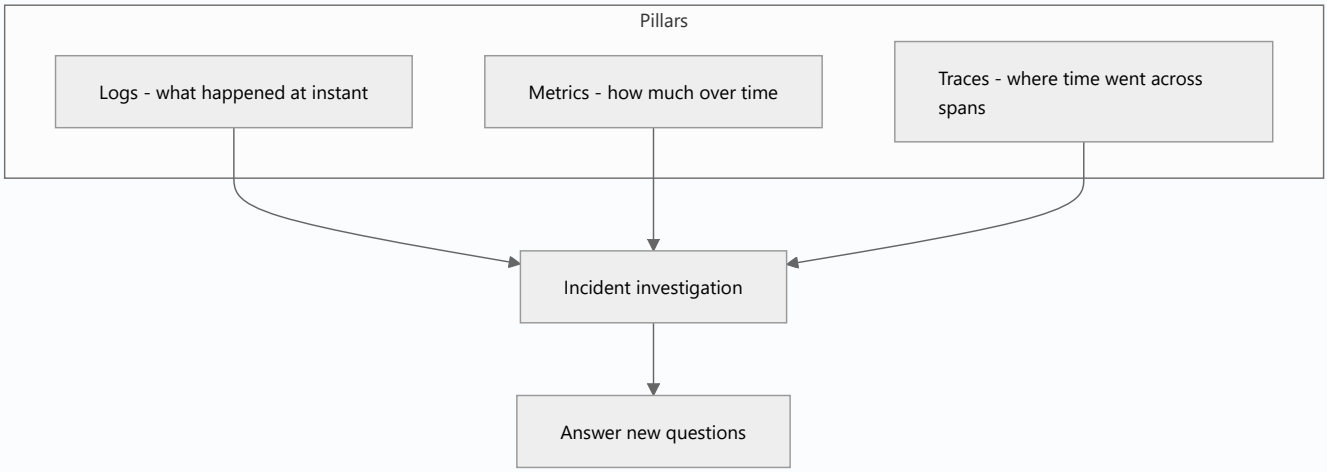
```
    (Object error, StackTrace stack) {
      unawaited(reportToCrashReporter(error, stack));
    },
  );
}

class _App extends StatelessWidget {
  const _App();
  @override
  Widget build(BuildContext context) => const SizedBox.shrink();
}
```

Usage tying it together:

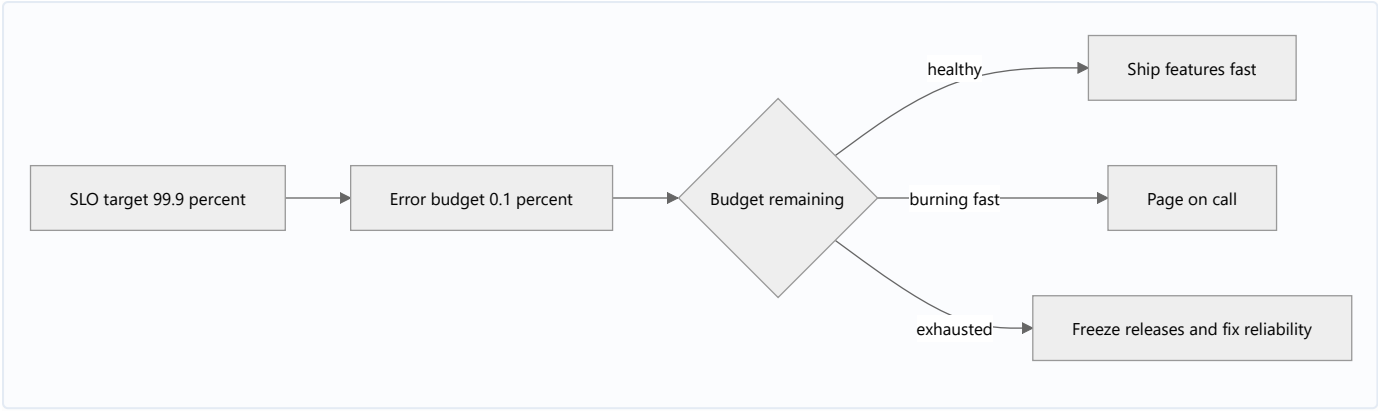
```
final StructuredLogger log = StructuredLogger(debugPrint as void Function(Map<String, Object?>));
final Counter checkoutStarted = Counter();

void onCheckoutTapped(String correlationId) {
  checkoutStarted.increment();
  log.log(LogLevel.info, 'checkout started',
      correlationId: correlationId, fields: <String, Object?>{'step': 'tap'});
}
```

Diagrams

The three pillars and what each answers:



SLO error-budget burn governing release pace:



Common Mistakes

Mistake	Why it hurts	Fix
Log everything at `info` / `debug`	Noise, storage cost, and PII leaks (emails, tokens, locations)	Log decisions and errors, not every value; scrub PII; use levels
Alert on causes (CPU high, pod restarted)	Fires constantly, often when users are fine → **alert fatigue**	Alert on **symptoms/SLO burn** the user feels
Vanity metrics (total signups ever)	Big number, no decision changes from it	Track actionable, rate-based SLIs
No correlation ID	Cannot stitch a user's flow across logs/services	Generate one per flow, propagate it (see Logging)
Dashboards nobody reads	False confidence; broken graphs go unnoticed	Fewer dashboards tied to SLOs; delete the rest
Averages only	Tail latency and rare hangs hide behind the mean	Track p95/p99 histograms
Unbounded label cardinality (userId as a metric label)	Time-series explosion, cost blowup, slow queries	Keep labels low-cardinality; put IDs in logs/traces

Best Practices

- **Instrument for questions, not for graphs.** Ask "what would I need to debug the next unknown outage?" and emit that.
- **One correlation ID per user flow**, propagated app → network header → backend, so all three pillars join.
- **Structured logs** (key/value maps), consistent field names, explicit levels — never free-text-only.
- **Metrics are low-cardinality and aggregatable**; identifiers belong in logs and traces.
- **Sample deliberately**: keep 100% of errors, sample the happy path. Make sampling decisions visible in the data.
- **Every alert links to a runbook** and maps to an SLO or user-visible symptom.

- **Scrub PII at the source** (before it leaves the device), not "later in the pipeline."
- **Crash-free users / sessions** is your headline mobile SLI; pair it with slow/frozen-frame rate.
- Treat observability as a **release gate**: no feature ships to a flow you can't observe.

Performance

Telemetry is not free — on mobile it competes for **CPU, battery, and the user's data plan**:

- **Hot-path cost**: never format strings or serialize JSON on the frame-build path when disabled; guard with level checks. Emit cheaply, process off-thread.
- **Batching** amortizes network/radio wake-ups; frequent tiny uploads drain battery (radio tail energy) more than periodic batches.
- **Sampling** caps trace/log volume and cost; head-sampling is cheapest.
- **High-cardinality metrics** are the classic cost bomb: each unique label combination is a stored series. `userId` / `requestId` as labels can create millions of series and dominate your bill and query latency.
- **Network/offline**: buffer to disk, upload on Wi-Fi where possible, respect background execution limits.
- **Symbolication** happens server-side, so release builds stay small and fast; only debug-symbol artifacts are uploaded.

Advantages

- Answer **unforeseen** questions without a redeploy — faster incident resolution (lower MTTR).
- Correlate across pillars to find root cause, not just "something is wrong."
- SLOs give an **objective, shared definition of "good enough."**
- Error budgets turn reliability-vs-velocity from an argument into a policy.
- Better product decisions from real-user data (network, device, region distributions).

Disadvantages

- **Cost**: storage, egress, and vendor bills scale with volume and cardinality.
- **Overhead**: CPU/battery/network on device; latency if done on the hot path.
- **Complexity**: three pillars, sampling, and correlation add moving parts.
- **Privacy risk**: telemetry is a PII leak waiting to happen if unscrubbed.
- **Noise if undisciplined**: too many signals is as blinding as too few.
- **Mobile blind spots**: sampled, delayed, and incomplete — you never see 100% of reality.

Interview Questions

🟢 **What is the difference between monitoring and observability?** Monitoring watches predefined signals for known failure modes (known-unknowns) via dashboards and alerts. Observability is a property of the system: its outputs are rich enough that you can ask *new* questions about internal state (unknown-unknowns) without shipping code. Monitoring is a subset/consumer of observability.

🟢 **Logs vs metrics vs traces — when do you reach for each?** Logs: discrete events with full detail — best for "what exactly happened at this moment," worst at aggregation and cost at volume. Metrics: aggregated numbers over time — best for trends, alerting, and cheap long retention, worst at explaining a single event and at high cardinality. Traces: causal span timelines — best for "where did the time/latency go across services," worst at long-term storage and usually sampled. Use metrics to detect, traces to localize, logs to explain.

🟢 **What is an error budget?** `1 - SLO`. If your SLO is 99.9% success, your error budget is 0.1% of events allowed to fail per window. It is a currency: while budget remains, ship features fast; when it burns too fast or exhausts, freeze feature releases and spend engineering on reliability. It converts reliability-vs-speed into an explicit, data-driven policy.

🟢 **Define SLI, SLO, SLA.** SLI is the measured indicator (e.g., % of requests under 300ms). SLO is the internal target for that SLI (e.g., 99.9%). SLA is the external, contractual promise to customers, usually looser than the SLO, with penalties. SLI is fact, SLO is goal, SLA is contract.

🟡 **Why alert on symptoms rather than causes?** Causes (high CPU, restarts) fire often without user impact, causing alert fatigue and desensitization. Symptoms (error rate up, latency SLO burning) map to what users actually feel, so every page is actionable. You investigate causes *after* a symptom pages you.

🟡 **What is cardinality and why does it matter for metrics?** Cardinality is the number of unique label-value combinations. Each combination is a stored time series. High-cardinality labels (userId, requestId, URLs with IDs) multiply series count, exploding cost and slowing queries. Keep metric labels low-cardinality; put unique identifiers in logs and traces.

🟡 **How do you capture uncaught errors in a Flutter app?** Wire `FlutterError.onError` for sync framework errors, `PlatformDispatcher.instance.onError` for uncaught async platform errors (return `true`), and wrap `runApp` in `runZonedGuarded` for zone-escaping async errors. Forward all three to the crash reporter. Add isolate error listeners for spawned isolates.

🟡 **What's a correlation ID and how does it work across a user flow?** A unique ID generated at the start of a logical flow and attached to every log/span/request it produces, propagated as a header to the backend. It lets you reconstruct one user's journey across app, network, and services from otherwise disconnected records.

🔴 **Why is mobile observability fundamentally harder than server?** You don't own the device or network; telemetry is sampled, delayed, buffered offline, arrives out of order and duplicated; you can't attach a debugger to a user's phone; battery/data constraints cap what you can send; and privacy law/PII rules limit what you may log. You reason from incomplete, lagging samples, not a live, complete stream.

🔴 **How does an error budget govern release velocity in practice?** Releases are gated on remaining budget. Healthy budget → normal or accelerated shipping. Fast burn → page and slow down. Exhausted → automatic feature freeze; only reliability fixes ship until the budget recovers over the rolling window. This makes reliability a shared, enforced constraint rather than a preference.

🔴 **What makes a good postmortem, and what is the "five whys"?** Blameless (focus on systems, not individuals), factual timeline, clear impact, contributing factors, and concrete action items with owners. "Five whys" is iteratively asking "why did that happen?" (~5 times) to move past the proximate cause to the systemic root — e.g., crash → null response → no schema validation → no contract test → no CI gate.

🔴 **Mitigation before diagnosis — why?** During an incident the priority is stopping user pain (roll back, feature-flag off, shed load), not understanding root cause. Diagnosis can take hours; users are hurting now. Stabilize first, investigate after, learn in the postmortem.

Senior Engineer Tips

- **Add the correlation ID before you think you need it.** Retrofitting it mid-incident is impossible.
- **Delete telemetry aggressively.** Unread dashboards and never-fired alerts are liabilities, not assets.
- **Test your alerts.** An alert that never fired is not "healthy" — it may be broken. Fault-inject to verify.
- **Log at the boundary.** Every network call and every external dependency gets a structured log with the correlation ID.
- **Keep a "golden signals" view**: latency, traffic, errors, saturation — plus crash-free users for mobile.
- **PII scrubbing is a code-review checklist item**, not a hope.
- **Write the runbook when you write the alert**, while you still remember what it means.

Architect Perspective

Observability is a **design requirement, not an afterthought**. At architecture time you decide: what correlation ID scheme, what field naming standard, what sampling policy, what the SLOs are, and which flows are "unobservable" (and therefore not allowed to ship). Bolting telemetry on

after launch produces exactly the blind spots that cause the outage you can't debug.

At the org level, the architect's leverage is **standards**: one logging schema, one tracing propagation format, one metrics naming convention, one crash reporter, one incident process. Fragmentation — every team inventing its own fields — makes cross-service correlation impossible precisely when you need it most.

Crucially, **SLOs govern release velocity**. The architect frames reliability not as "be careful" but as an error-budget policy wired into the release pipeline (cross-link: Delivery and Release). This aligns product pressure and reliability into one negotiable currency, owned by the whole org rather than argued per-launch.

Summary

Monitoring answers questions you already knew to ask; observability lets you ask new ones from the system's outputs. The three pillars — **logs** (discrete events), **metrics** (aggregated time-series), **traces** (causal span trees) — complement each other: metrics detect, traces localize, logs explain. Discipline keeps it useful: structured logs with levels and a correlation ID, low-cardinality metrics, symptom-based alerting tied to SLOs, and every alert backed by a runbook. **SLI/SLO/SLA** define "good enough," and the **error budget** ties reliability to release pace. Incident response is a practiced discipline: mitigate before you diagnose, run blameless postmortems, ask the five whys. On mobile the hard truth is that you don't own the device: telemetry is sampled, delayed, offline-buffered, and privacy-constrained, so crash-free-users, ANR/hang rates, and jank become your headline signals. Design observability in from the start — it is an architectural requirement, not a patch.

Revision Notes

- Monitoring = known-unknowns; observability = unknown-unknowns.
- Pillars: logs (event stream), metrics (time-series), traces (span tree).
- Error budget = 1 − SLO; governs how fast you ship.
- Alert on **symptoms / SLO burn**, not causes. Every alert → runbook.
- Global error net: `FlutterError.onError` + `PlatformDispatcher.instance.onError` + `runZonedGuarded`.
- Mobile: sampled, delayed, offline-buffered, PII-limited; crash-free-users + ANR + jank.
- Keep metric labels low-cardinality; put IDs in logs/traces.
- Incident: mitigate → diagnose → blameless postmortem → five whys.

Practice Questions

1. Write one sentence distinguishing monitoring from observability without using the words "known" or "unknown."

2. You have a rising p99 latency but flat averages — explain why and which pillar you'd use next.
3. Given SLO 99.95% over 30 days, compute the error budget in minutes of allowed failure.
4. Name three things you must **never** put in a log for a Flutter app in a regulated market.
5. A metric labeled by `sessionId` costs a fortune — explain the mechanism and the fix.
6. Draft a symptom-based alert for checkout and contrast it with a cause-based one you'd delete.
7. Explain why mobile telemetry can show fewer errors than actually occurred.

Coding Questions

1. Wire a global error handler to a reporter. Acceptance criteria:

- `FlutterError.onError`, `PlatformDispatcher.instance.onError` (returns `true`), and `runZonedGuarded` around `runApp` are all set.
- Every path forwards `(error, stack)` to a single `reportToCrashReporter` function with a `fatal` flag.
- No unawaited-future lint warnings; null-safe; compiles clean under `flutter analyze`.

2. Define 3 SLIs + SLOs for a Flutter app. Acceptance criteria: choose three user-visible indicators (e.g., crash-free sessions, checkout success rate, cold-start p95), each with a measurable SLI definition, a numeric SLO target, a measurement window, and the resulting error budget. Justify why each is a symptom users feel.

3. Add a correlation ID to logs. Acceptance criteria:

- A single ID is generated at flow start and threaded through every log in that flow.
- The ID is attached as a field in structured output and as an outbound HTTP header.
- Two different concurrent flows never share an ID; demonstrate with a short test.

Mini Project

Instrument a small Flutter feature end-to-end.

Pick one feature (e.g., "add to cart → checkout"). Deliver:

1. **Structured logs** — key/value events at flow boundaries with `LogLevel` and a per-flow `correlationId`, PII-scrubbed. (Reuse the `StructuredLogger` above; see Remote Logging & Observability for shipping them.)
2. **One metric** — a counter `checkout_started_total` and a histogram `checkout_duration_ms`, both low-cardinality (label by result: success/failure only).
3. **Crash capture** — the three-handler global error net forwarding to a reporter (Crashlytics/Sentry stand-in; see Crash Reporting).
4. **One SLO** — "99% of checkouts complete under 4s over a rolling 7 days," with its error budget computed.

5. **One alert rule** — page when the checkout success SLI burns budget at >2× the sustainable rate for 30 minutes; attach a one-paragraph runbook (first action: feature-flag checkout to the previous flow — mitigate before diagnose).

Deliverable: a short doc + working code showing a single checkout producing correlated logs, an incremented metric, a recorded duration, and — when you throw a test error — a captured crash event carrying the same `correlationId`.

See also: Monitoring · Logging · Remote Logging & Observability · Crash Reporting · Delivery and Release