Controlled Change

Architecture, Reliability, and Evolution for Mobile Systems

MIKE SALARI

Contents

Contents of the full book. Chapter and figure numbers in this sample are those of the full book.

Part I. Decision Discipline

1. Architecture Is Controlled Change IN THIS SAMPLE

2. Why Mobile Is Different

3. Characteristics, Budgets, and Evidence

4. The Economics of Abstraction

5. Decisions, ADRs, and Revisit Triggers

6. Architecture Across Product Stages

Part II. Features and State

7. From Screen to Feature

8. Dependency Direction Without Ceremony

9. Domain-Driven Mobile Boundaries

10. State Ownership and Unidirectional Flow

11. The Pattern Laboratory

12. Navigation, Lifecycle, and Restoration

13. Dependencies, Composition, and Injection

Part III. Data Under Failure

14. Swift Concurrency as Architecture

15. Kotlin Coroutines as Architecture

16. Process Death, Background Work, and Time IN THIS SAMPLE

17. Data Authority and Repository Design

18. Local Data, Cache, and Freshness

19. Offline-First Is a Product Decision

20. Building a Sync Engine

21. Conflicts, Idempotency, and Reconciliation

22. Schema Evolution and Version Skew

23. Real-Time and Event-Driven Mobile

Part IV. Scale, Code, and Teams

24. Feature-Based Modularization

25. Designing the Module Graph
26. When Modularization Goes Wrong
27. Team Topologies and Ownership
28. Build Architecture and Feedback Loops
29. Mobile Platform Engineering
30. Cross-Platform with Escape Hatches
31. SDK Architecture and Compatibility
32. Migrating Legacy Mobile Systems

Part V. Production Systems

33. Testing the Failure Model
34. Performance as an Architecture Characteristic
35. Observability, Reliability, and Field Truth
36. Security, Privacy, and Trust Boundaries
37. Accessibility, Adaptivity, and Device Diversity
38. Release Architecture
39. Feature Flags, Experiments, and Kill Switches IN THIS SAMPLE
40. Server-Driven UI and Remote Experiences

Part VI. AI-Native

41. The AI-Native Mobile Stack
42. On-Device, Cloud, and Hybrid Routing
43. Context, Memory, RAG, and Multimodality
44. Agentic Apps, Permissions, and Tools
45. Offline AI, Model Lifecycle, and Evaluation

Part VII. Staff Practice

46. Architecture Reviews That Work
47. Fitness Functions, Governance, and Debt
48. Influence, RFCs, and Architecture Roadmaps
49. Incident-Led Architecture
50. Mobile System Design Interviews
51. Designing Systems That Can Be Replaced

Part VIII. Casebook

52. Atlas: From Prototype to Platform

53. The Offline Field Operation
54. The Global Checkout
55. The Messaging System
56. The Regulated Finance App
57. The Cross-Platform Rewrite
58. The AI Trip Operator
59. The Mobile Platform Program
60. The Agent in the Engineering Loop IN THIS SAMPLE
61. The Imposed Decision

Appendices

Appendix A. Mobile Architecture Decision Canvas
Appendix B. ADR Library
Appendix C. Architecture Review Rubric
Appendix D. Migration Playbooks
Appendix E. Testing and Failure Catalog
Appendix F. Performance and Reliability Budgets
Appendix G. Security and Privacy Checklist
Appendix H. Staff+ Interview Bank
Appendix I. Glossary
Index

Preface: Architecture Begins After the Diagram

Most architecture material begins in a clean room. The requirements are legible. The network responds. The database is empty. The team agrees. No customer is running a version from eighteen months ago. No release manager is deciding whether a two-percent regression is acceptable on Friday afternoon.

Mobile systems do not live there.

They live on constrained devices, under hostile scheduling, across operating-system generations, behind app-store review, beside unreliable networks, with users who kill the process at inconvenient moments. They are changed by several teams at once. They depend on backends that evolve on a different clock. Their releases cannot be recalled like server deployments. Their architecture is judged by code clarity and also by startup time, battery, accessibility, privacy, build feedback, rollback speed, and the quality of evidence available during an incident.

This book puts those facts at the center of architecture. Most treatments leave them for the appendix.

It discusses Clean Architecture, MVVM, MVI, Redux, TCA, VIPER, RIBs, Workflow, feature modules, Domain-Driven Design, reactive systems, and dependency injection. None is presented as a destination. Each is examined as a response to pressure: state pressure, ownership pressure, failure pressure, team pressure, delivery pressure, or change pressure.

The running system is **Atlas**, a marketplace and field-operations product used by travelers, merchants, and local operators. Atlas supports discovery, booking, payment, messaging, media, location, offline work, background synchronization, feature flags, progressive rollout, and an AI planning assistant. It is intentionally too complex for one fashionable diagram and realistic enough to expose architectural trade-offs.

The iOS and Android implementations remain native where platform mechanics matter. A backend and contract layer make version skew visible. Kotlin Multiplatform, React Native, and Flutter examples test the boundaries of shared code. Simulators inject duplicate events, reordered writes, stale clients, process death, and model failure. The purpose is to make decisions observable, not to advertise a stack.

The thesis is simple:

> *A good mobile architecture lets a team change a product repeatedly, under unreliable networks, constrained devices, evolving platforms, multiple client versions, organizational growth, and production failure, without losing control of correctness, delivery speed, user experience, or cost.*

That is a demanding definition, and most of what follows is about meeting it.

What kind of book this is

This is a book about architectural judgment. It is not an implementation encyclopedia, a framework manual, API documentation, or a catalog of universal patterns. Where it names a platform API or a framework behavior, it does so to show the decision, and the current details belong to official documentation at the moment you decide. What the book teaches is how to reason about mobile systems under change, failure, scale, production constraints, organizational pressure, and releases that stay on devices for years.

Who this book is for

You can already build a mobile app. You have shipped features, fixed the crash, argued about MVVM in a pull request. What you are reaching for now is the judgment that decides which boundary is worth its cost, which migration can run without stopping the release train, and which decision will still be defensible in three years when the person who made it has left.

This book is for Staff and Principal mobile engineers, mobile architects, platform leads, and the engineering managers who fund their decisions. It assumes implementation competence and begins where architectural judgment becomes the harder problem.

Who should not read it

Engineers learning their first architecture will find this book moves too fast and skips the tutorial. That is deliberate. There is no code walk-through here that teaches dependency injection from scratch, no folder template to copy, and no framework crowned as the answer. If that is what you need, this is probably not the right starting point. This book is about the decisions underneath the code, and it is unapologetic about the level it is pitched at.

A note on how it is written

I have tried to write the book I wanted when I was the one holding the decision and the pager at the same time. Where the book makes a consequential claim, I try to state what evidence would challenge it. Where I state a trade-off, I name what it costs, not only what it buys. Where a large company's practice is instructive, I have kept the lesson and dropped the logo. A famous name proves nothing about your product; the useful check is whether the pressure that justified their choice exists in yours. Copy the reasoning, not the diagram.

How to argue with this book

Atlas is fictional. It combines pressures common to mobile systems, and because I wrote it, it will tend to agree with me. Treat it as a test bench rather than proof. Three habits make the book useful even where you disagree with it.

First, check the pressure before the recommendation. Every recommendation in these chapters starts from a pressure: a lost response, an old client, a team that cannot change alone. If your system does not have that pressure, the recommendation does not apply, however convincing the Atlas story is.

Second, run the failure, not the argument. Most chapters describe a failure you can produce on purpose: kill the process, drop a response after the server commits, send an old payload to a new backend, switch a flag off while the app is offline. The companion code does this for Atlas. An afternoon doing it to your own app will tell you more than any chapter.

Third, treat the numbers as illustrations. Budgets, thresholds, and percentages in the book are marked illustrative because they are. The methods behind them, how to measure, what to compare, which authority to trust, are meant to carry over. The values are not.

Where the book is wrong for your system, the counterexamples and “when not to” sections are the places to start, because I wrote them to mark the edges of my own advice.

About the Author

Mike Salari is a staff engineer and tech lead who has spent fifteen years building and shipping mobile systems on iOS and Android. He has designed the platforms several teams build on, run the architecture reviews where those decisions are made or unmade, and carried the pager when they went wrong in production.

He writes at **salari.dev** for engineers who have outgrown the tutorial shelf and now carry decisions that outlive their own tenure.

1. Architecture Is Controlled Change

Architecture is often introduced as structure: layers, arrows, modules, and boxes. Structure matters, but it is an incomplete unit of analysis. A beautifully separated application that cannot be migrated, observed, released safely, or understood by the team is well illustrated, but it is not well architected.

A more useful question than *What pattern does this app use?* is this one:

> *How cheaply and safely can this product absorb the next important change?*

That change may be a new business rule, a database migration, an operating-system behavior, a privacy constraint, a second platform, an acquisition, a tenfold increase in engineers, or a model that now runs partly on-device. Architecture determines how much unrelated surface must move, how much evidence is available before release, how quickly harm can be contained, and how reversible the decision remains.

A model of architectural value

Treat the following model as a framing device, not as accounting. Its purpose is to correct abstraction worship:

> **Architectural value =**
>
> $$\frac{\text{Changeability} \times \text{Reliability} \times \text{Delivery capability} \times \text{Product leverage}}{\text{Cognitive load} + \text{Coupling} + \text{Operational risk} + \text{Migration cost}}$$

Adding a protocol may improve substitutability, and every reader now has one more concept to hold. Splitting a module can isolate ownership while slowing the build. A local database improves continuity and brings reconciliation with it. A reducer makes transitions deterministic, and it can also turn a trivial screen into an event vocabulary nobody wanted. Every architecture move buys one property by spending something else.

Atlas begins with a search screen and a booking button. Two engineers can understand the whole application. A single feature target, a modest networking layer, and direct platform state are the rational choice at that size. Six months later, bookings must survive process death, hosts need offline operations, and payments cannot be duplicated when the response is lost. The original structure did not become morally wrong. The pressure changed.

Five placements that define a mobile architecture

Most consequential decisions place one of five things.

Boundary. Which changes are allowed to propagate? A payment provider belongs behind a boundary because its contracts, certification, and failure semantics differ from the product language. Formatting a subtitle usually does not.

Ownership. Who may mutate a piece of state, schema, module, or release control? Multiple writers without a reconciliation policy are not flexibility; they are an undeclared distributed system.

Authority. Which representation wins when local intent, cached data, backend state, and optimistic UI disagree? The repository pattern does not answer this. An authority policy does.

Lifetime. How long may work or state exist: view lifetime, feature lifetime, process lifetime, device lifetime, or account lifetime? Many concurrency bugs are lifetime mistakes disguised as threading mistakes.

Evidence. How will the team know whether a decision works in the field? A boundary that cannot be measured is a belief. Release architecture, telemetry, and tests turn it into an engineering hypothesis.

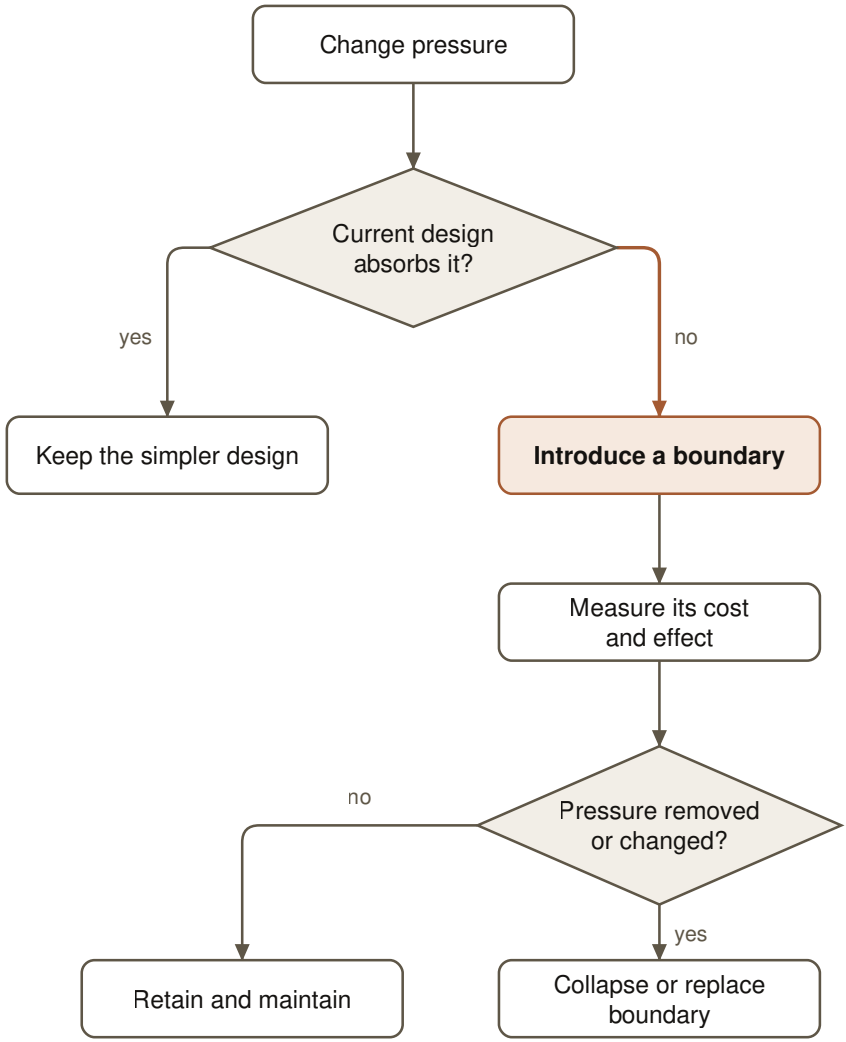

Figure 1.1 Introduce a boundary only when change pressure requires it

Architecture is temporal

A class diagram captures one instant. Production architecture includes before and after:

- the path from old storage to new storage;
- the period in which old and new clients share a backend;
- the interval between enabling a feature and knowing whether it is healthy;
- the time a queued operation can remain unresolved;
- the date a compatibility bridge must be removed;
- the conditions that cause an ADR to be revisited.

This temporal view changes how designs are evaluated. A rewrite proposal is not complete when it describes the destination; it must describe coexistence, migration, rollback, and deletion. The same goes for a cross-platform proposal: an estimate of shared lines says little until it covers native escape hatches, ownership, debugging, release coupling, and the cost of returning to platform code.

The controlled-change test

Before approving an architecture, walk through six changes:

1. A product change: inventory becomes reservable for ten minutes.
2. A failure change: the network response is lost after the server commits.
3. A scale change: three teams edit the booking journey.
4. A platform change: background execution becomes more restricted.
5. A policy change: a region requires different data retention.
6. A replacement change: the payment SDK must be removed within one quarter.

For each, identify the affected modules, data migrations, contracts, release controls, observability, and owner. The exercise exposes accidental coupling faster than debating pattern names does.

What architecture is not

Architecture is not maximum separation; separating everything erases the signal that a boundary is important. A universal folder template does not supply it either, because identical directories can hide radically different authority and lifecycle models. Frameworks encode useful choices, but they cannot supply product context. Nor is architecture the Staff engineer's private taste. It is a shared operating model, and everyone who changes the system pays for it.

TRADE-OFFS

Controlled change favors reversible decisions and pressure that someone has named. It may appear less elegant than adopting one comprehensive doctrine. The reward is proportionality: a small feature can remain small, while a safety-critical flow earns stronger boundaries, state machines, persistence, and release control.

FAILURE MODES

Teams fail when they treat architecture as a one-time phase, optimize for diagram purity, copy a large-company solution without its pressure, or keep abstractions after the reason for them disappears. Another failure is local cleanliness with system-level fragility: impeccable feature layers attached to unversioned APIs, unsafe releases, and invisible background work.

MIGRATION PATH

Begin by documenting current authority, ownership, lifetime, and evidence for one critical journey. There is no need to redraw the entire app. Find the change that is currently expensive, introduce the smallest boundary that contains it, add a measurable outcome, and write a removal condition. Repeat where pressure proves the need.

ARCHITECTURE REVIEW QUESTIONS

- What change is this design making cheaper or safer?
- What does it make more expensive?
- Where are state, authority, and lifetime placed?
- How will old and new behavior coexist?
- What field evidence will validate the choice?
- Under what condition should the design be simplified or replaced?

STAFF ENGINEER PERSPECTIVE

The Staff engineer makes the cost of change legible, aligns each boundary with organizational responsibility, and leaves the team with a decision it can operate after the author moves on. Producing more architecture than that is not part of the job.

16. Process Death, Background Work, and Time

Mobile applications exist in discontinuous time. The process disappears, clocks move, connectivity returns later, and scheduled work receives a brief opportunity. Designs that assume a continuous in-memory executor eventually lie to users.

Memory is a cache of intention

Anything essential after process death must have a durable representation before the UI claims acceptance. For Atlas FieldOps, an operator captures proof of completion in a basement with no network. The app must persist the work record, media references, operation identity, dependency ordering, and user-visible status. A coroutine or task is not the durable representation.

The experience distinguishes “saved,” “sending,” “needs attention,” and “confirmed.” Calling all four “success” creates dangerous retries and support confusion.

Background schedulers provide opportunity, not guarantees of timing

iOS background tasks and Android WorkManager operate under platform policy, which weighs battery, network, user behavior, and resource pressure. Architecture must tolerate delayed execution and short windows.

A scheduler should wake a durable processor and never hold the only copy of the work. The processor:

1. queries eligible operations;
2. acquires a lease with expiration;
3. verifies dependencies and account scope;
4. performs one bounded attempt;
5. records acknowledgement, unknown outcome, retry time, or terminal failure;
6. releases resources;
7. asks for future scheduling if work remains.

Foreground resumption can run the same processor, and correctness should not depend on which trigger happened to supply execution time.

Leases make interruption recoverable

Marking an operation “in progress” without a lease can strand it forever after process death. A lease records owner identity and expiration, so a new processor may reclaim work after expiry. Use monotonic elapsed time for in-process durations and durable wall-clock instants for persisted coordination, while recognizing that device clocks can change.

Backend leases or idempotency ledgers may be required for cross-device effects. A local lease only prevents concurrent processors on one storage authority.

Time is a dependency

Time influences token expiration, price validity, retry, feature rollout, cache freshness, conflict resolution, and model manifests. Inject a clock into policy. Distinguish:

- wall-clock time for human and server timestamps;
- monotonic time for elapsed durations;
- server-authoritative time where client clock manipulation matters;
- logical or sequence clocks for ordering distributed events;
- timezone/calendar policy for user experience.

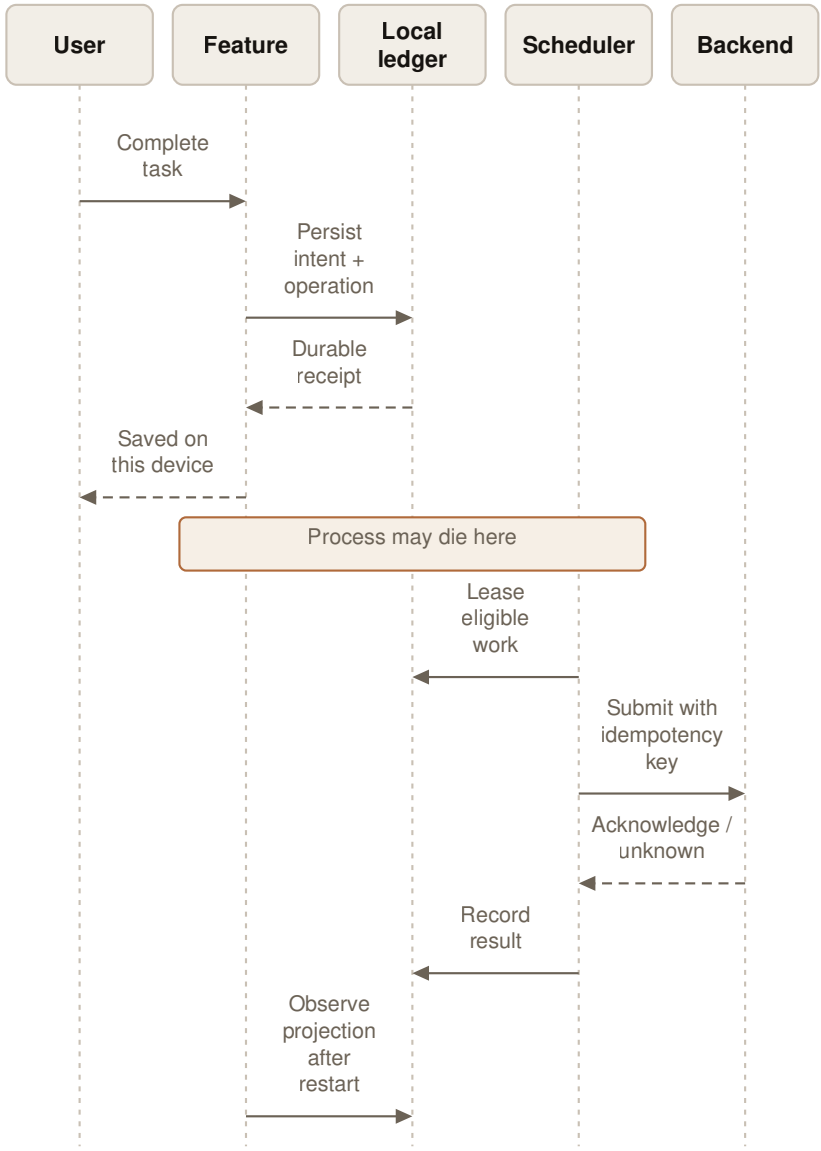

Figure 16.1 Intent is saved before work runs, so it survives process death

Never use last-writer-wins solely from untrusted device wall clocks for consequential conflicts.

Deadline is not timeout

A timeout limits how long a caller waits. A deadline states when a business operation ceases to be acceptable. A booking quote may expire at a server-provided instant; a network attempt can time out earlier and be retried. A background upload may continue after the screen's observation timeout.

Represent them separately; that enables truthful states: "We are still checking whether the reservation completed" rather than "Request failed."

Restore from durable facts

On cold start, reconstruct in layers:

1. open and migrate local stores;
2. establish account and encryption scope;
3. load durable operations and projections;
4. resolve expired leases;
5. restart eligible processing;
6. parse the requested route;
7. render a state that reflects known authority;
8. reconcile when connectivity permits.

Replaying every UI action is the wrong default. Restore minimal feature inputs and derive current state from durable facts. Action replay can re-trigger effects unless the architecture is specifically event-sourced and idempotent.

Media and large artifacts

Large uploads need file ownership. Copy or move selected media into application-controlled storage before accepting the operation; external picker URLs may become inaccessible. Record content hash, size, media type, encryption, upload parts, and cleanup policy. Upload completion and backend attachment are separate states, and orphan cleanup has to respect outcomes that are still unknown.

One operation through three interruptions

An operator finishes an inspection in a basement with no signal, takes two photos, and taps Complete. Follow that single operation from the tap to cleanup.

1. **Acceptance.** The feature copies both photos into app storage, records their hashes and sizes, and writes the work record and operation `op-71` in one local transaction. The screen says “Saved on this device” only after the commit. If the app dies before the commit, the screen never claimed anything, and the task is still open when the operator returns.
2. **First attempt.** Forty minutes later she walks upstairs. A connectivity callback wakes the processor, which leases `op-71` for five minutes and starts uploading the first photo in parts. The system suspends the app halfway through the second part. The lease is still held, so no other processor touches the operation.
3. **Process death.** Overnight the system reclaims the app’s memory. No code runs. The lease simply passes its stored expiry.
4. **Recovery.** The next morning she opens the app. Startup follows the restore order above: the store opens, account scope is established, and the processor finds `op-71` with an expired lease. It reclaims the operation and asks the upload service which parts already arrived. Only the missing part is sent.
5. **Unknown outcome.** The attach request times out after the server has committed it. The ledger records `unknown`, and the screen says “Checking whether this was received.” The next attempt carries the same idempotency key, the server returns the original attachment, and the ledger moves to `confirmed`.
6. **Cleanup.** Only now do the local photo copies become eligible for deletion, under the retention policy for evidence.

Each step depends on a fact written down earlier. None of them depends on memory surviving, on the scheduler running at a particular time, or on the device clock matching the server’s. Step 3 has one weak point: the lease expiry is a device wall-clock instant, so a clock moved backward can delay reclaiming the work. The cost is a delay. Correctness is unaffected, because the idempotency key protects the effect.

Kill the process on purpose

A restart test has to end the process. Recreating a view model in the same process proves nothing about durability. A useful harness:

- drives the operation to a named transition: persisted, leased, part uploaded, attach sent, response lost;
- terminates the process at that point, for example with `adb shell am kill` on a backgrounded Android app, or with `terminate()` followed by `launch()` in an iOS UI test;
- relaunches cold and checks both what the user sees and what the ledger holds;
- checks that the backend recorded at most one logical effect for the operation ID.

Run it once for every transition in the state machine. The companion sync simulator covers the two cases that matter most, with the process dying after the server committed and before the request left the device, in `ProcessDeathTests`. The first must reconcile by status without resending. The second must resend the same operation exactly once.

User-initiated versus opportunistic work

The product should prioritize work the user is actively waiting for while preserving fairness and battery. A queue can classify interactive, durable-critical, normal, and maintenance work. A poison operation should never block unrelated items, and large media transfers should respect network constraints and the user's cellular-data settings.

TRADE-OFFS

Durable intent improves continuity and brings schema, encryption, migration, and cleanup work with it. Background scheduling helps work finish eventually; it cannot promise when. Leases support recovery at the cost of clock and ownership complexity. Rich restoration improves the experience, and a poorly scoped one preserves stale or sensitive state.

FAILURE MODES

Queues kept in memory are the first thing to look for. After that come workers assumed to complete, device wall time used as truth, side effects replayed on restoration, inaccessible media URLs, and timeouts labeled as failures. Background retry that ignores account logout or user cancellation belongs on the list too.

WHEN NOT TO PERSIST

Ephemeral search requests and disposable UI animations should die. Persistence is warranted for user intent, costly work, legal/audit state, or outcomes whose uncertainty matters. Most taps never need to become durable events.

MIGRATION PATH

Pick one operation currently vulnerable to interruption. Define a durable record and state machine. Persist before UI acknowledgement, add a foreground processor, then integrate platform scheduling. Add restart tests at every transition and explicit cleanup for terminal states.

ARCHITECTURE REVIEW QUESTIONS

- What remains true after immediate process death?
- Is the scheduler a trigger or the owner of work?
- How are leases reclaimed?
- Which clock is used for each decision?
- How are timeout, deadline, cancellation, and unknown outcome distinguished?
- What happens on logout and device storage pressure?

39. Feature Flags, Experiments, and Kill Switches

A feature flag is a distributed policy decision executed inside a fleet of old and new clients. It can reduce release risk, support experiments, and create a state space no team can reason about. The difference is architecture and governance.

One bag for every boolean hides the differences that matter. The purpose determines the lifetime, default, audience, and evidence.

Classify flags by job

TYPE	PURPOSE	EXPECTED LIFETIME	SAFE DEFAULT
Release flag	separate deployment from exposure	days to weeks	old proven path
Experiment	compare product hypotheses	bounded by analysis plan	control assignment
Operational kill switch	disable harmful capability	long-lived control, rarely active	safest functional state
Permission/ entitlement	represent server-authorized access	ongoing	deny until authorized
Migration flag	select old or new implementation or data path	migration window	rollback-capable path
Configuration parameter	tune bounded behavior	ongoing with schema/version	embedded validated value
Development flag	local or testing convenience	non-production	absent from production path

An entitlement is not merely a rollout flag. A client-side flag cannot authorize paid or privileged behavior; the backend remains authoritative.

Evaluate through a typed policy layer

Avoid string keys spread through features:

```
public struct BookingPolicy: Sendable, Equatable {
    public let useNewCheckout: Bool
    public let maxOfflinePendingHours: Int
    public let allowOnDeviceSummary: Bool
}
```

Snippet 39.1. Illustrative fragment. Related tested code: ios-atlas `ReleasePolicy.swift`, which evaluates typed feature rules, and `ReleasePolicyTests.swift`.

A policy adapter can translate remote configuration, client capability, account entitlement, experiment assignment, and embedded defaults into a validated domain policy. The feature consumes the policy, not the vendor SDK.

Benefits:

- unknown or missing values receive explicit defaults;
- policy can be snapshotted for one journey;
- tests enumerate meaningful combinations;
- the vendor can be replaced;
- sensitive targeting stays at the correct boundary;
- telemetry records the decision, not every raw config field.

Snapshot versus live evaluation

Some values may change during a session; others must remain stable for a workflow. Checkout should not switch implementation or price behavior halfway through because a background config refresh arrived.

Choose an evaluation scope:

- **per application launch** for broad experience configuration;
- **per account or session** for entitlement and cohort;
- **per journey** for transactional consistency;
- **per action** only when immediate operational control is required and safe;
- **server-authoritative** for high-impact policy.

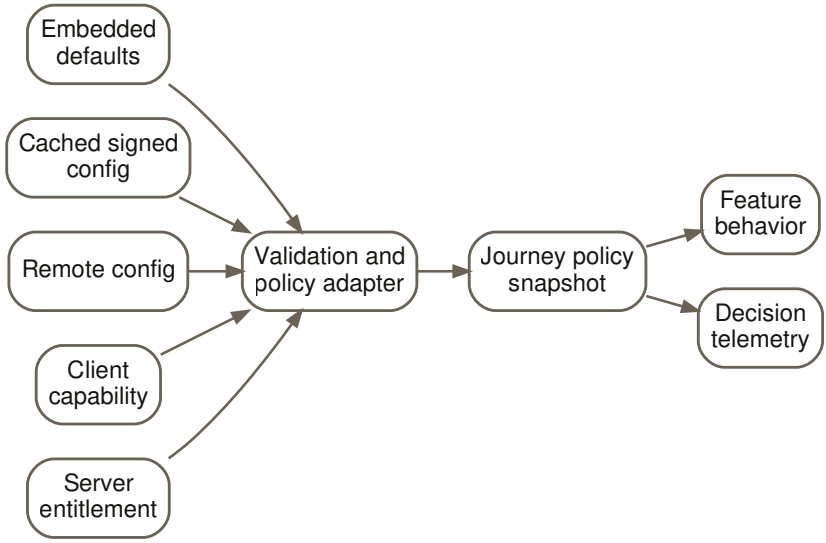

Figure 39.1 All flag inputs pass through one policy adapter into a journey snapshot

Record the evaluated policy version, or the relevant decisions, with the operation so support can explain behavior after configuration changes.

Defaults and offline behavior

Each flag must answer the configuration failures listed in Chapter 38, including a fresh install that never connected and a cache belonging to another account. Define:

- embedded default;
- cache scope and expiry;
- signature or integrity validation where risk warrants;
- behavior when the value is absent, unknown, or out of range;
- whether the last-known value is safe;
- account and privacy boundaries;
- emergency override path;
- client capability floor.

For a new checkout release flag, default to the proven old flow. For a security kill switch, the default may disable the vulnerable action. For a legally required disclosure, stale cached text may be unsafe; the feature may need a connection or a minimum version.

Experiments need architecture discipline

An experiment should define:

- hypothesis and primary outcome;
- guardrail metrics such as reliability, performance, support, and accessibility;
- unit of randomization;
- inclusion, exclusion, and consent requirements;
- assignment persistence;
- exposure event semantics;
- interference between devices and accounts;
- analysis window and stop rule;
- cleanup decision.

A flag evaluation is not an exposure. Record exposure when the user experiences the treatment; otherwise the analysis is biased.

Keep the control path tested and compatible. Experiments that alter a backend contract, local schema, or queued operation meaning need version-aware design, not just two UI branches.

Kill switches must be boring and proven

A kill switch should:

- be independent of the failing feature path;
- have a safe embedded or cached behavior;
- reach enough of the installed fleet quickly;
- be controlled by least privilege and audited;
- be observable by release and incident teams;
- have a tested manual procedure;
- avoid bricking unrelated capability;
- work with old supported clients when possible.

Exercise kill switches in non-production and in controlled production cohorts. A control used only during an incident is likely to be misunderstood when it is needed.

Flag debt is architecture debt

Each temporary flag needs metadata:

```
key
owner
purpose/type
created date
expected expiry or revisit trigger
safe default
affected modules and contracts
removal issue
experiment or rollout dashboard
```

Automate expiry warnings and prevent new dependencies on flags marked for removal. Deleting a flag includes:

- choosing the permanent branch;
- removing dead code and tests;
- deleting remote configuration;
- cleaning analytics dimensions;
- collapsing schemas or migrations when safe;
- updating documentation and runbooks.

A permanent two-path architecture doubles testing and incident ambiguity.

Targeting and privacy

Target only with data that is necessary, lawful, and governed. Avoid transmitting sensitive traits to a third-party flag service when evaluation can occur through coarse server-assigned cohorts or local capability.

Treat configuration as potentially untrusted input. Validate ranges and enum cases. Never let a remote value supply arbitrary class names, URLs, SQL, prompts, or tool permissions without allow-listed interpretation.

State-space testing

Flags multiply the release compatibility matrix of Chapter 38. Pairwise testing can cover ordinary independent combinations, but critical interactions deserve explicit scenarios:

- old client + new backend + flag off;
- new client + old-compatible backend + flag on rejected by capability;
- config unavailable + cached stale policy;
- account switch changes entitlement;
- operation created under old policy resolves under new policy;
- experiment treatment plus accessibility setting;
- kill switch during active workflow;
- rollback after local schema has already migrated.

Testing every arbitrary combination is hopeless. Reduce the state space instead: scope flags, snapshot policy, and delete flags promptly.

TRADE-OFFS

Flags reduce rollout exposure and add configuration state, testing, and governance. Experiments accelerate learning; poorly designed ones compromise experience or privacy. Kill switches contain incidents and concentrate privileged control. A typed policy layer adds translation, which is what protects features from vendor and config churn.

FAILURE MODES

String keys spread throughout the codebase are the first sign of trouble. Then come live flag changes mid-transaction, unsafe defaults, entitlements enforced only on the client, exposure logged at evaluation, permanent migration flags, remote values interpreted as code, and kill switches never exercised.

MIGRATION PATH

Inventory flags and classify each one's job, owner, default, and age. Wrap one critical domain in a typed policy object. Delete the oldest safe flag. Add expiry enforcement, configuration-failure tests, and a kill-switch drill. Extend to experiments with explicit exposure and guardrails.

SCORECARD

Score every flag in the inventory once a quarter. A flag that fails two rows goes on the removal list.

QUESTION	HEALTHY ANSWER
What kind of flag is it?	Release, experiment, operational, or permission, recorded in the inventory
When does it disappear?	A date or a condition, with an owner
Where is it evaluated and snapshotted?	One named scope, such as journey start
What happens on first launch, offline, or with invalid configuration?	A tested embedded default
Does the backend still enforce authorization?	Yes, independent of the flag
Are guardrails and exposure defined for experiments?	Exposure logged at the decision point, guardrails set in advance
Can the switch contain harm while the guarded code fails?	Proven in a drill

60. The Agent in the Engineering Loop

Atlas now has nine mobile teams. Most engineers use coding agents every day. Agents write view models, update tests, bump dependencies, and draft migrations. The platform team is asked a question that sounds simple: can agents merge some of their own changes without a human review, so engineers spend less time approving routine work?

Chapter 44 treated an agent as part of the product, acting for a traveler. This case treats agents as part of the engineering system, acting on the code. The same principles hold. Authority stays with something that can be held accountable, the model proposes, and evidence decides. What changes is the thing at risk: not a booking, but the installed app on several million devices.

The pressure

The platform team collects a month of data before proposing anything.

- About 60 percent of merged pull requests were written mostly by an agent.
- Median review time for those pull requests is twelve minutes, and most of that time goes to reading changes that turn out fine.
- Two incidents in the quarter came from agent-written code that passed review: a background upload moved into a screen-scoped task, and a renamed field in a shared payload model that broke old clients.
- Engineers report that they approve agent changes faster than they used to approve each other’s, and that they understand them less.

All figures in this chapter are illustrative. The shape is common: review has become the bottleneck, and it is also not catching the failures that matter.

Separate the question into task classes

The first decision is to stop asking about “agent changes” in general. The team sorts six months of agent pull requests into classes by what they touch and how reversible the damage is.

CLASS	SHARE OF AGENT PRS	DAMAGE IF WRONG	MACHINE-CHECKABLE?
Dependency bumps	18%	Build or runtime break, reversible next release	Mostly
Presentation	31%	Visible defect, reversible	Mostly, with snapshots
Feature logic	27%	Wrong state, sometimes data	Partly
Concurrency and lifecycle	9%	Lost or duplicated work on device	Only with restart tests
Contracts and persistence	8%	Old clients and stored data	Only with fixtures from old versions
Build, signing, release	7%	Every user or every developer at once	Poorly

Two classes, dependency bumps and presentation, are both common and checkable. They are the only candidates for reduced review. The other four keep human review, and two of them, contracts and the release path, also get a deny list: agents may draft changes there, but cannot edit signing material, store metadata, release branches, or production flag files at all.

Bound before you trust

Before measuring anything, the team removes the ways an agent could do damage that no review would catch in time.

- Each agent task runs in its own worktree on its own branch.
- Deny rules, enforced by the agent tool and again by a pre-push hook, cover signing, store metadata, release branches, production configuration, generated code, and secrets. Deny wins over allow.
- Shell commands run in a sandbox with network access limited to the package registries and the internal build cache.
- Hooks make the always-true rules executable: both platforms compile, lint passes, tests for touched modules pass, and no forbidden path changed. An agent cannot report “done” with a failing check without the run saying so.
- Project guidance files are trimmed to one page per module and moved next to the code they describe.

None of this makes the agent better. It makes certain failures impossible, which is a stronger property than making them unlikely.

Separate the writer from the judge

The second change is structural. The agent that writes a change no longer gets the only automated opinion on it. A second model from a different provider reviews every agent pull request with reviewer instructions and no access to the writer’s session. Its blocking comments go back to the writer as tasks. A human sees the pull request after one adversarial round.

Qualify one class with evidence

The team proposes auto-merge for dependency bumps first. They write the method down before running it, following Chapter 45: the evaluation is a release gate, and the thing being qualified is the arrangement of agent, checks, and routing rule, not the agent.

1. **Success rule.** A bump is good if the build and all tests pass, no source file other than the lockfile and version catalog changed, and no revert or incident is traced to it within two weeks.
2. **Cases.** 260 past bumps, split once at random into 130 for tuning and 130 held out. Repeated attempts at the same bump count as one case.
3. **Routing rule, tuned on the tuning set.** Auto-merge when every rung is green, the reviewer model raised no blocking comment, and the diff touches only allowlisted files. Target: at least 97 percent of auto-merged bumps good, tuned to 99 percent on the tuning set to leave a margin.
4. **Freeze.** The rule is written into the decision record before anyone looks at held-out results.
5. **Qualify.** On the held-out set the rule auto-merges 96 bumps. All 96 are good. The one-sided 95 percent lower bound on reliability is about 96.9 percent.

That is just below the 97 percent target. With zero failures in 96 cases, the evidence can rule out a failure rate above roughly 3 percent, and no lower. The team does not adjust the target or the rule to make the result pass. They record “insufficient evidence”, keep human review, and keep collecting cases. Three months later they have 280 new held-out bumps. The same frozen rule auto-merges 200 of them, and one of those fails. The lower bound is now about 97.7 percent, above the target, and auto-merge for dependency bumps is switched on. The switch is a flag, and the policy, its lineage, and its triggers are an ADR.

Presentation changes take longer, because screenshots catch layout defects but not every accessibility regression. The team adds an accessibility audit to the ladder before even starting the count for that class.

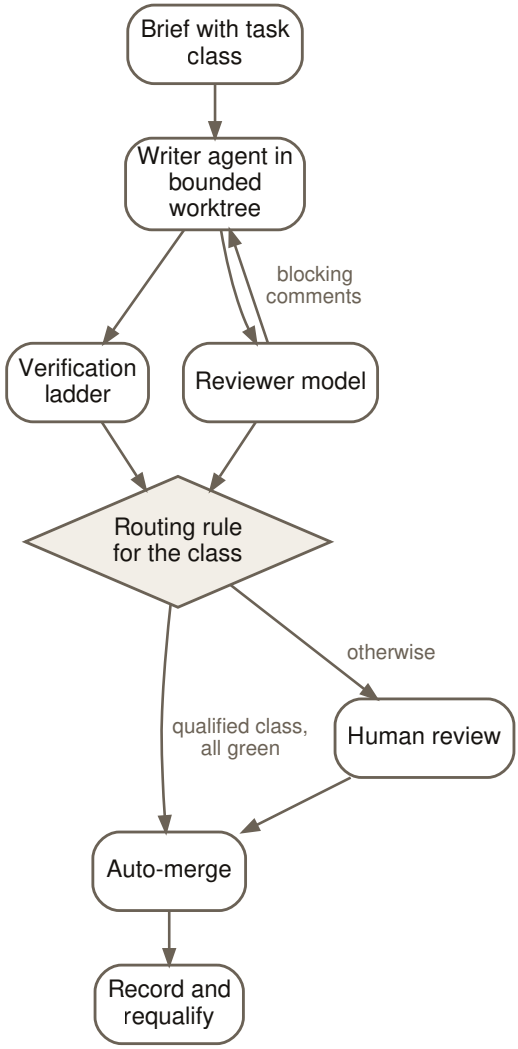

Figure 60.1 Atlas separates the agent that writes from everything that judges

Write down what ends the qualification

The ADR names the conditions that reopen the decision:

- the writing or reviewing model, or the agent tool, changes version;
- permissions, hooks, or the verification ladder change;
- auto-merged bumps cause more than one revert in a rolling month;
- the dependency set changes in kind, for example a new platform or a native SDK with its own build steps;
- six months pass.

When the writing model was upgraded three months later, auto-merge for the class was switched off by the same flag, and the qualification was rerun on fresh held-out cases before switching it on again. That took four days. Nobody argued about whether the new model was “better”; the evidence answered it.

The incidents, revisited

The two incidents that started the work are now checks on the ladder.

- **The screen-scoped upload.** Concurrency and lifecycle changes now require the restart test from Chapter 16: start the operation, kill the process, relaunch, and assert that the operation resumed and the server recorded one effect. The agent that moved the upload would have failed it.
- **The renamed field.** Contract changes now require fixtures from the oldest supported client, as in Chapter 22. A rename fails the old-client decode test before review.

Neither check is about agents. Both protect against a human making the same mistake, which is why they were worth building. Agents made the gap more expensive, because they produce more changes for the same amount of review.

What the humans do now

The review burden fell for two classes and stayed the same for four. Engineers did not get the time back for free; they spend part of it on what the loop cannot do.

- **Explain before approving.** Approving an agent change requires the reviewer to state, in the pull request, what the change does and what would break if it were wrong. Reviews took slightly longer and caught more.
- **Write the first one by hand.** New boundaries, new patterns, and incident fixes under time pressure keep human authorship. Agents follow a pattern well once a person has set it.
- **Read for the agent's habits.** Reviewers learned to look first for duplicated helpers, convention drift, weakened tests, and files outside the brief.
- **Own the memory and the guidance.** Instruction files, agent memory, and shared skills are reviewed each quarter like any other code, because stale guidance keeps steering new code.

Context is part of the system

One failure did not fit any class. For a month, agents kept writing new code against the legacy networking client, although the migration to the new client had finished. Nothing in the code told them to. The cause was a note in the shared agent memory, written during the migration, that said new endpoints should use the legacy client "until the cutover". The cutover happened; the note stayed. Every session loaded it, and every session followed it.

The fix was ordinary cache hygiene. Memory entries now point to the decision record instead of restating it, so they cannot drift from it. Entries carry an owner and a review date. Anything an agent read from an issue comment, a web page, or a dependency's documentation is never written into memory as an instruction, because text aimed at the agent can arrive that way. Tool servers, plugins, and shared skills are treated as dependencies: allowlisted sources, pinned versions, and a switch that turns each one off for every engineer at once.

When less review is the wrong goal

Not every team should run this program. A team of four on one app, where every engineer reads every change anyway and review takes minutes, gains little from qualification machinery. Its review is not the bottleneck, and the case set it could build would be too small to qualify anything at a useful target. For that team the right investment is the bounds and the checks, which cost little and prevent the worst failures, and nothing more.

The program also fails if it is used to justify a number decided in advance. If leadership wants “80 percent of agent code merged without review” by a date, the evidence will either support it or not, and a method that bends to the date is not a method. Atlas wrote down, before starting, that “insufficient evidence” was an acceptable outcome for every class, and that the classes touching signing, contracts, and stored data would not be candidates at all.

Staff exercise

Leadership asks to extend auto-merge to feature logic, because it is the largest class by volume. Write the response. Decide what success rule would have to hold, which checks the class lacks today, how many held-out cases the team would need to rule out a 2 percent failure rate, and which part of the class, if any, could be qualified first. Then name the condition under which you would recommend never qualifying it.

WHAT THE CASE TEACHES

Agents change how much code a team produces, not what makes mobile code safe. The controls that let Atlas trust agents with more work are the same ones the rest of this book argues for: bounds that make dangerous actions impossible, checks that machines can run, a judge separate from the author, and decisions qualified on evidence and reopened when the ground moves. A team that has those can let agents do more. A team that does not will find out through its users.

Inside the full book

- 61 chapters in eight parts, written in pressure order rather than pattern order.
- Atlas, a marketplace and field-operations product, is the running system through the whole book.
- 10 chapters on data under failure: concurrency, process death, sync, conflicts, idempotency, schema evolution, and version skew.
- 5 chapters on AI on mobile: routing, context and memory, agentic apps, model lifecycle, and evaluation.
- A casebook of 10 cases, from Atlas: From Prototype to Platform to The Imposed Decision.
- 9 appendices of working tools, including 12 example ADRs and 10 migration playbooks.
- 74 diagrams.

The three editions

Book, the standard edition: the complete book as PDF and EPUB.

Pro: the book plus the Atlas reference system: runnable iOS, Android, Kotlin Multiplatform, and backend code with contract schemas.

Extended: everything in Pro, plus React Native, Flutter, and the sync simulator that injects duplicate events, reordered writes, process death, and model failure.

BUY THE FULL VERSION

Controlled Change

Architecture, Reliability, and Evolution for Mobile Systems

You have read 4 of the 61 chapters. The full book has the other 57.

All three editions are on one page; you pick the edition before you pay.

BUY ON GUMROAD

mikesalari.gumroad.com/l/controlled-change

ABOUT THE BOOK

salari.dev/books/controlled-change

Mike Salari
Staff mobile engineer · 500+ technical interviews from the hiring side