

Controlled Change

Architecture, Reliability, and
Evolution for Mobile Systems



MIKE SALARI

Controlled Change

Architecture, Reliability, and Evolution for
Mobile Systems

Mike Salari salari.dev

The architecture is not the diagram you begin with. It is the system of decisions that determines how safely you can change what already exists.

Copyright

Copyright 2026 Mike Salari. All rights reserved.

salari.dev

No part of this book may be reproduced, distributed, or transmitted in any form without the prior written permission of the author, except for brief quotations in a review.

Product and company names are the property of their respective owners. The casebook and the Atlas reference system are fictional. They combine common engineering pressures so that trade-offs can be discussed without exposing any organization's confidential systems.

The companion code is a teaching system. It demonstrates architectural mechanisms and failure handling. It is not a substitute for security review, legal review, accessibility testing, privacy assessment, production load testing, or platform compliance work in your own product.

Dedication

For the engineers who inherited the diagram after the people who drew it had left.

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 not only by code clarity but by startup time, battery, accessibility, privacy, build feedback, rollback speed, and the quality of evidence available during an incident.

This book treats those facts as the center of architecture rather than 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 not to advertise a stack. It is to make decisions observable.

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 definition is demanding.

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, because the point was never that a famous team did it. The point is whether the pressure that justified their choice exists in your product. Copy the reasoning, not the diagram.

How to Read This Book

The book is written in pressure order rather than pattern order.

Part I establishes a decision discipline. Part II follows a feature as state, behavior, navigation, and boundaries become difficult. Part III assumes the network fails and the process dies. Part IV adds teams, modules, build systems, shared platforms, and legacy code. Part V moves architecture into production: tests, performance, reliability, security, accessibility, and release control. Part VI treats AI as a new client/server boundary with its own budgets and threat model. Part VII addresses Staff+ practice. Part VIII is a casebook in which several architectures can be defensible, provided their assumptions are explicit. It ends with a decision the architect did not get to make.

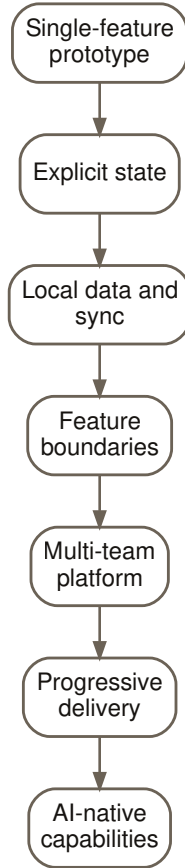
This book has a companion, the Atlas engineering system: runnable iOS, Android, Kotlin Multiplatform, and backend code, contract schemas, and a simulator that injects the failures architecture is supposed to survive. The Pro edition includes the native, Kotlin Multiplatform, backend, and contract code; the Extended edition adds React Native, Flutter, and the simulator. Both are available at salari.dev/books. The code is not a line-by-line tutorial. It is evidence. Read a chapter; open the matching project, run the tests, then alter an assumption. Disable connectivity. Duplicate a mutation. Delay an acknowledgement. Change a contract version. Expire a feature flag. Force the model router offline. Good architecture becomes clearer when it is stressed.

The **core path** is Parts I through V plus Chapters 41, 42, 46, 49, 52, and 54. The **Staff+ path** is the complete book. The **implementation path** pairs every part with the companion projects and the standalone decision artifacts in the toolkit.

Do not attempt to adopt everything. The book repeatedly argues against that behavior. Take the decision tools, identify your pressures, and pay only for the boundaries you need.

The Running System: Atlas

Atlas begins as a small travel marketplace and grows into a multi-region mobile platform.



How Atlas evolves across the book

A customer can discover local experiences, create an itinerary, reserve inventory, pay, message a host, store tickets, and continue essential journeys offline. A host can manage availability, confirm arrivals, capture evidence, and reconcile work performed without connectivity. Operations teams use controlled rollouts and remote configuration. An AI assistant proposes plans, summarizes conversations, and can prepare, but never silently execute, high-impact actions.

Atlas uses the same domain vocabulary across platforms:

- **Identity** owns accounts, sessions, and authorization context.
- **Catalog** (the Discovery feature) owns discoverable experiences and availability projections.
- **Itinerary** (the Trip Planning feature) owns customer intent and local planning.
- **Booking** owns reservation state and payment coordination.
- **Messaging** owns conversations and delivery state.
- **FieldOps** owns offline host workflows.
- **ReleaseControl** owns flags, experiments, and kill switches.
- **Assistant** owns AI context, model routing, policies, and tool proposals.

The vocabulary stays stable while implementations change. Stable product language around replaceable technical mechanisms is one of the book's recurring design moves.

Contents

PART I. DECISION DISCIPLINE	12
1. Architecture Is Controlled Change	13
2. Why Mobile Is Different	18
3. Characteristics, Budgets, and Evidence	23
4. The Economics of Abstraction	29
5. Decisions, ADRs, and Revisit Triggers	34
6. Architecture Across Product Stages	39

PART I

Decision Discipline

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 not well architected. It is well illustrated.

The useful question is not *What pattern does this app use?* It is:

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 =

Changeability × Reliability × Delivery capability × Product leverage

Cognitive load + Coupling + Operational risk + Migration cost

Adding a protocol may improve substitutability and increase cognitive load. Splitting a module may isolate ownership and slow the build. A local database may improve continuity and introduce reconciliation. A reducer

may make transitions deterministic and 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 rational. 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.

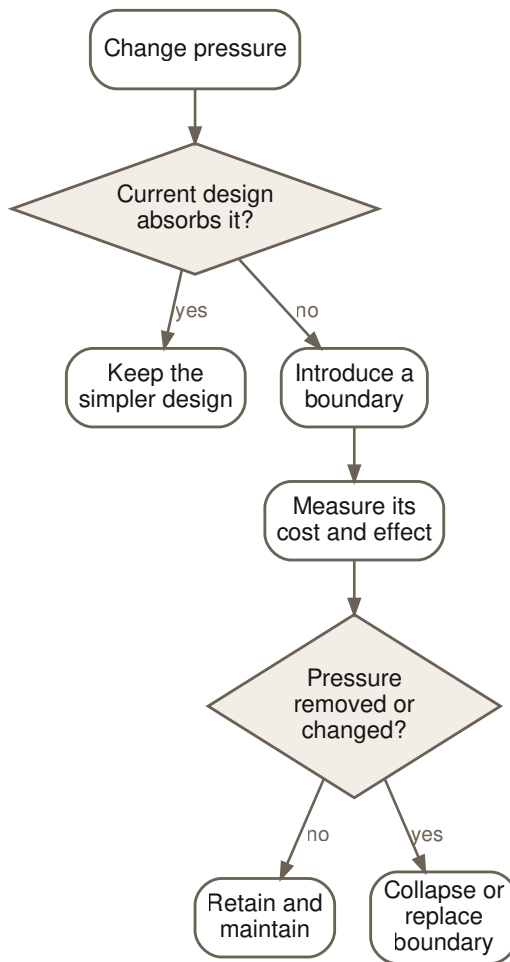


Figure 1.1 A model of architectural value

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.

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. A cross-platform proposal is not

complete when it estimates shared lines; it must describe 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. It is not a universal folder template; identical directories can hide radically different authority and lifecycle models. It is not framework adoption; frameworks encode useful choices but cannot supply product context. It is not the Staff engineer's private taste. Architecture is a shared operating model whose cost is paid by everyone who changes the system.

TRADE-OFFS

Controlled change favors reversible decisions and explicit pressure. 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. Do not 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's job is not to produce the most architecture. It is to make the cost of change legible, align each boundary with organizational responsibility, and leave the team with a decision that can be operated after the author moves on.

2. Why Mobile Is Different

A mobile application is a distributed-system participant that runs on a device the team does not control, under a scheduler it does not control, across a network it does not control, and through a release channel it cannot instantly reverse. Treating it as a small web frontend hides the forces that should shape its architecture.

The client is durable, old, and intermittently alive

A server can often be deployed again. A mobile binary may remain active for months. Users postpone updates, devices run older operating systems, stores stage distribution, and emergency fixes still require adoption. At any moment the backend may serve several app versions with different schemas, feature sets, and assumptions.

The process is also disposable. The user backgrounds the app; the operating system reclaims it; a scheduled task receives seconds rather than minutes; a notification opens a deep link into a state the app has not reconstructed. Memory is not durable truth. Every important journey must declare what survives view recreation, feature recreation, process death, logout, and device replacement.

The network has more states than online and offline

A request can fail before reaching the server, after reaching it but before committing, after committing but before the response arrives, or after the response arrives but before local persistence. Radio transitions consume energy. Captive portals produce connectivity without usefulness. Mobile networks reorder latency and availability by location, carrier, and movement.

`try/catch` is therefore not a reliability model. A mutation needs an operation identity, an acknowledgement model, a retry policy, and a reconciliation strategy. Reads need a freshness contract. Streaming needs catch-up. Authentication refresh needs single-flight ownership. A spinner is not an architecture for uncertainty.

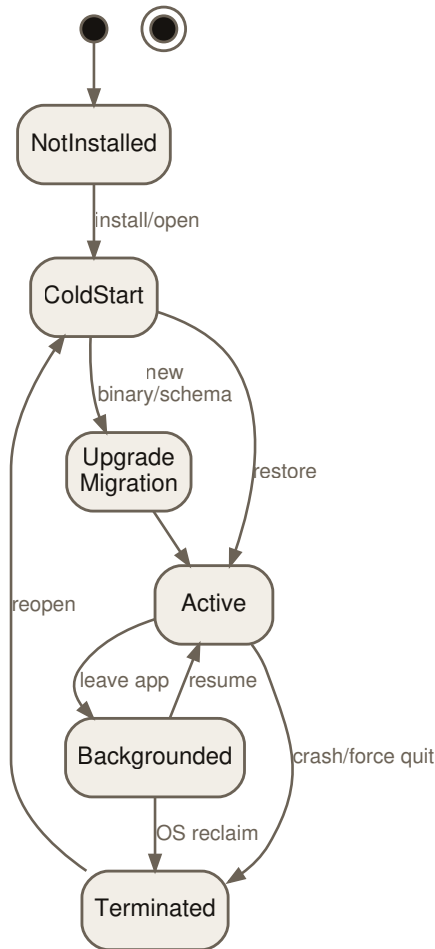


Figure 2.1 The client is durable, old, and intermittently alive

Distribution is part of the system

Server teams often separate runtime architecture from deployment. Mobile teams cannot. A decision that requires all clients to update is usually not deployable. A feature without a remote disable path may force a binary release while harm continues. A database migration without rollback behavior can turn a release into permanent local corruption. App review, phased release, store metadata, entitlements, signing, and platform policy are architectural constraints because they determine how change reaches production.

Consider Atlas. A new booking flow depends on an API field, a payment SDK, a local pending-operation table, and a feature flag. The architecture includes the release sequence:

1. deploy backward-compatible backend support;
2. ship dormant client capability;
3. validate schema migration and telemetry;
4. enable for employees;
5. widen by cohort and platform version;
6. monitor business correctness and technical health;
7. retain a server-side kill switch;
8. remove the old path only after the compatibility window closes.

The code path and the rollout path are one design.

The device is a constrained, privacy-sensitive edge

CPU, memory, storage, battery, thermals, bandwidth, and disk I/O are budgets. The camera, location, contacts, health data, secure hardware, and notifications are privileged capabilities. Background work is rationed. Model files can add hundreds of megabytes. A design that is acceptable on a workstation may make a phone hot, slow, inaccessible, or invasive.

These constraints produce architecture questions:

- Which computation belongs on-device, at the edge, or in the cloud?
- Which data must never leave the device?
- Which local artifacts need encryption, expiration, or user-visible deletion?
- Can the feature degrade when low-power mode, thermal pressure, or a metered connection is present?
- Does the app remain useful with a large text size, screen reader, keyboard, foldable layout, or intermittent sensor access?

Platform behavior is not an implementation detail

Swift concurrency isolation, Android coroutine scopes, SwiftUI and Compose state, background schedulers, lifecycle callbacks, secure enclaves,

keystores, app attestations, and accessibility trees affect where boundaries can be placed. A platform abstraction that erases these mechanics can make code look portable while making failures harder to reason about.

This is why the book shows iOS and Android side by side without pretending they are identical. The principles can align (single ownership, structured lifetime, durable intent, explicit authority) while the mechanisms remain native.

Mobile systems include humans and organizations

A release manager deciding whether to halt rollout, an on-call engineer diagnosing a crash cluster, a support agent resolving an unknown booking state, and a privacy reviewer inspecting data flow are runtime participants. Their tools and evidence are part of the architecture.

Team boundaries likewise become dependency boundaries. When ten teams share a mutable core, naming it `common` solves nothing. When a platform team owns a golden path without adoption metrics or escape hatches, it becomes a gatekeeper. Mobile architecture has to connect technical boundaries with ownership and delivery.

TRADE-OFFS

Native integration preserves platform capability and can duplicate concepts. Shared abstractions reduce repeated business logic and can conceal lifecycle differences. Durable local state improves continuity and creates migration and reconciliation obligations. Remote control reduces release risk and creates configuration, security, and flag-debt risks.

FAILURE MODES

Common failures include assuming the latest client, treating memory as durable state, retrying non-idempotent work, coupling a backend migration to immediate adoption, interpreting reachability as successful connectivity, putting authorization in the client, and measuring only crashes while users experience hangs, stale data, or duplicate effects.

WHEN NOT TO GENERALIZE

Do not introduce a sync engine for a read-only brochure. Do not build a platform abstraction around one platform-specific capability merely to

make directories symmetrical. Background execution is unnecessary when foreground recovery satisfies the product. Mobile constraints demand respect, not maximum machinery.

ARCHITECTURE REVIEW QUESTIONS

- What happens on an old binary after the backend changes?
- Which state survives process death?
- What does a lost response mean for every mutation?
- How is a harmful feature disabled without a new binary?
- What resource and privacy budgets constrain the design?
- Which platform mechanics must remain visible?

STAFF ENGINEER PERSPECTIVE

The phone is not the whole product, and it is not a trusted server. It is an occasionally connected replica, an experience surface, a privileged edge computer, and a long-lived release artifact.

3. Characteristics, Budgets, and Evidence

“Fast,” “reliable,” “secure,” and “maintainable” are aspirations until a scenario gives them shape. Architecture work improves when adjectives become budgets with an owner, a measurement, and a response when the budget is exceeded.

Start with quality scenarios

A useful quality scenario contains six elements:

1. **Source:** who or what applies pressure;
2. **Stimulus:** the event;
3. **Environment:** the operating conditions;
4. **Artifact:** the affected part of the system;
5. **Response:** the intended behavior;
6. **Measure:** the threshold.

For Atlas checkout:

When a signed-in customer taps Confirm on a degraded mobile network, the booking subsystem durably records the intent before acknowledging success to the UI, produces at most one reservation, resolves an unknown response without a second charge, and shows a truthful status within the agreed recovery window.

That sentence is architecture. It implies local persistence, operation identity, backend idempotency, state transitions, reconciliation, telemetry, and support semantics. “Checkout must be reliable” implies nothing testable.

Build a characteristic portfolio

Architecture characteristics compete. Lower startup time can conflict with eager initialization. Stronger offline behavior can conflict with immediate global consistency. More module isolation can conflict with build simplicity. More analytics can conflict with privacy and energy.

Create a portfolio rather than a wish list:

CHARACTERISTIC	ATLAS SCENARIO	INITIAL BUDGET	EVIDENCE
Startup	cold launch to usable itinerary	p75 ≤ 1.2 s on reference class	field trace
Booking correctness	confirmed intent creates duplicate reservation	zero tolerated	ledger + reconciliation
Offline continuity	operator completes required steps without network	100% of critical workflow	scenario simulation
Release safety	harmful capability can be stopped	≤ 10 min after decision	kill-switch drill
Accessibility	core booking path at largest text and screen reader	no blocked task	automated + manual audit
Build feedback	changed feature produces local verification	p75 ≤ 90 s	CI/build telemetry
Privacy	precise location retained beyond need	zero by policy	data inventory + deletion test

The numbers are examples, not universal standards. The practice is what matters: a budget forces a trade-off into the open.

Architecture budgets are spendable

A performance budget is consumed by startup work. A cognitive-load budget is consumed by concepts a contributor must hold. A compatibility budget is consumed by schema divergence and old-client support. A failure budget is consumed by retries, fallbacks, and degraded modes. A release-complexity budget is consumed by flags, staged dependencies, and branches.

No feature spends without consequence. Suppose an AI itinerary assistant adds a 900 MB optional model, background downloads, embeddings, local retrieval, cloud fallback, and model telemetry. The product value may justify

it, but the architecture review must account for storage, startup scanning, network policy, consent, thermal pressure, privacy, and deletion. “It runs on-device” is not a complete benefit statement.

Use fitness functions carefully

A fitness function is an automated or routinely evaluated check that protects an architectural property. Examples include:

- no feature module may depend on another feature’s implementation;
- startup initialization must remain below a traced budget;
- every remote action type must map to an allow-listed local command;
- all persisted operation schemas require migration fixtures;
- every experiment flag has an owner and expiration date;
- dependency cycles fail the build;
- sensitive fields are absent from general telemetry payloads.

Automation is valuable when it protects a property cheaply. It becomes harmful when the rule is a proxy nobody can explain. “Every feature has exactly three layers” is not a fitness function; it is a template enforced by machinery.

Field evidence outranks laboratory confidence

Tests establish known properties under controlled conditions. Benchmarks detect regressions in representative environments. Production telemetry reveals device diversity, version skew, network geography, and human behavior. None is sufficient alone.

For startup, use deterministic benchmarks to catch code regressions and field percentiles to understand reality. For sync, use a simulator to inject duplicates and reordering, contract tests to validate API expectations, and reconciliation metrics to detect unresolved operations. For accessibility, combine static checks with assistive-technology testing and completion outcomes.

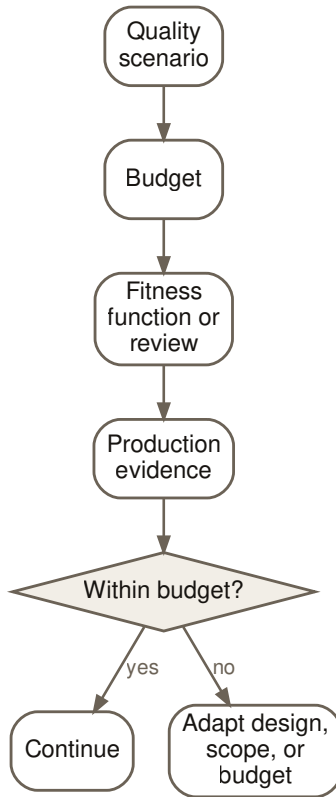


Figure 3.1 Use fitness functions carefully

Evidence should preserve the user's privacy. Prefer aggregate, purpose-limited measurements; minimize payloads; restrict access; define retention; and test deletion. Observability is not permission to record everything.

Make exceptions explicit

A budget without an exception process either becomes theater or blocks legitimate work. A waiver should state:

- which budget is exceeded;
- why product value justifies it;
- affected cohorts and devices;
- mitigation;
- owner;

- expiration or revisit trigger;
- evidence required to retain the exception.

Temporary exceptions need deletion pressure. Otherwise architecture debt gains official paperwork and permanent residency.

TRADE-OFFS

Precise budgets improve alignment but can create false certainty. Percentiles hide segments unless the team slices by relevant device, version, geography, and journey. Strict gates prevent regressions and can slow experiments. The answer is not weaker evidence; it is tiered rigor based on consequence.

FAILURE MODES

Teams measure what tooling exposes rather than what users experience, optimize averages, set goals with no owner, or change thresholds after failures to preserve a green dashboard. Another failure is treating a security or correctness invariant as an error budget. Some outcomes (double charging, authorization bypass, irreversible data loss) require prevention and containment, not an allowed percentage.

MIGRATION PATH

Choose one critical journey. Write three quality scenarios, assign initial budgets from current evidence, instrument the path, and run a review after several releases. Add automated gates only after the signal is understood. Expand by risk, not by dashboard enthusiasm.

ARCHITECTURE REVIEW QUESTIONS

- Which three characteristics dominate this decision?
- What are the concrete scenarios and thresholds?
- Which characteristic is being sacrificed?
- What evidence exists before and after release?
- Who owns a breach, and what action follows?
- Is the metric safe and privacy-preserving?

STAFF ENGINEER PERSPECTIVE

The Staff contribution here is making invisible quality requirements negotiable without making them optional. The artifact is not a large scorecard. It is a small set of consequential scenarios that connects product intent, technical design, release control, and field evidence.

4. The Economics of Abstraction

Abstraction is a loan. It provides leverage now and creates servicing cost for every future contributor. The debt is not inherently bad. The mistake is borrowing without naming the return.

What an abstraction costs

A boundary can cost:

- a new concept and vocabulary;
- navigation between files and modules;
- mapping between representations;
- build graph edges;
- indirection during debugging;
- mocking or test-double infrastructure;
- migration and compatibility work;
- versioning and ownership;
- performance overhead;
- a narrower set of engineers who can change it confidently.

The cost compounds. One repository interface is small. Repeating a presentation/domain/data template across seventy trivial features creates hundreds of types whose purpose is structural compliance. The organization then pays in code review, build time, search, onboarding, and mechanical migrations.

The absence of a boundary also costs. Direct use of a volatile payment SDK across the app makes replacement expensive. Shared mutable state reduces initial ceremony and increases defect radius. A monolithic target accelerates early development and later serializes teams. Architecture is the choice of which cost to pay and when.

The boundary test

Before adding an abstraction, ask four questions.

Volatility: Are the two sides likely to change for different reasons? A pricing rule and a JSON decoder are. A view-specific date string and its text label usually are not.

Consequence: What happens if change leaks? Payment, identity, local migrations, and remote commands justify stronger containment than a decorative card.

Multiplicity: Are there real alternative implementations or consumers? “We might replace this someday” is weak. iOS, Android, background work, widgets, and tests may create real multiplicity.

Coordination: Will different teams or release clocks own the sides? An organizational seam can justify a technical contract even when the code is currently simple.

Duplication can be cheaper than coupling

Two similar pieces of code are not automatically one concept. The traveler itinerary and host task list may both sort dated items. Sharing a `TimelineEngine` before their rules stabilize couples independent domains. Small duplication preserves autonomy. Shared code becomes attractive when the knowledge, invariants, and change cadence are truly shared.

The inverse also holds: teams sometimes duplicate high-consequence logic to avoid coordination. Reimplementing payment amount calculation or authorization policy in several clients is not healthy independence. The trusted backend should own the invariant; clients may own projections and validation for experience, never final authority.

Price module boundaries with graph economics

A module can reduce incremental build scope, enable ownership, and constrain dependencies. It can also add configuration, resource boundaries, generated interfaces, and dependency graph complexity. The unit of modularization should reflect change and ownership, not screens.

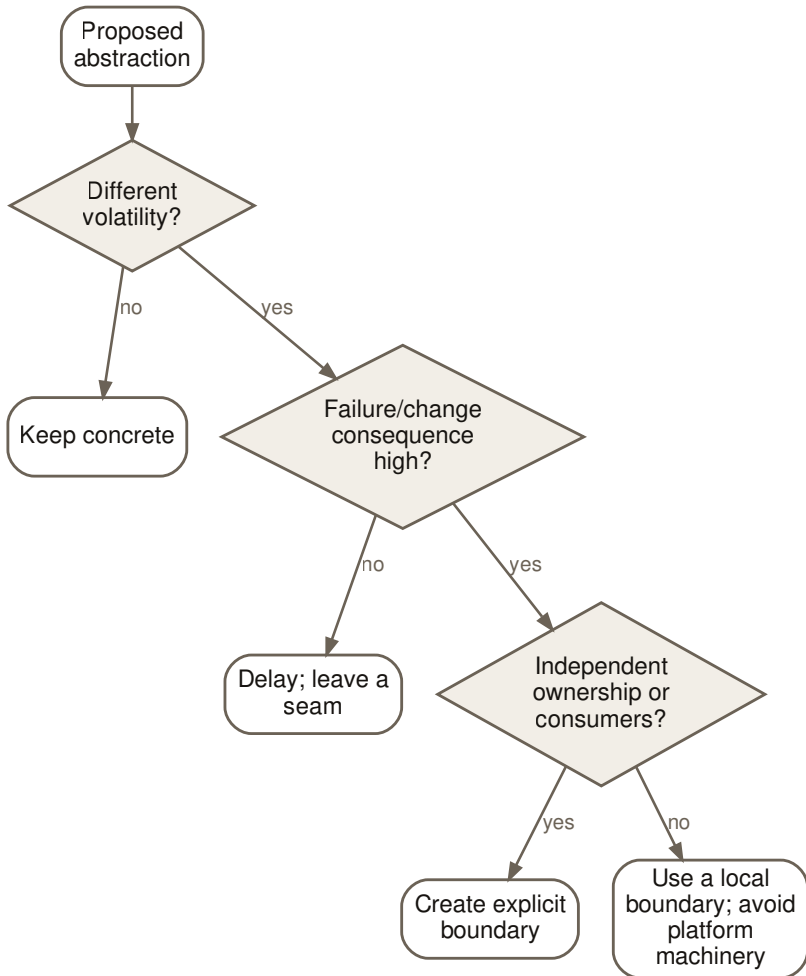


Figure 4.1 The boundary test

Useful module economics include:

- **fan-in:** how many modules depend on this one;
- **fan-out:** how many dependencies it imports;
- **change coupling:** which modules change together in history;
- **build criticality:** whether it lies on many compile paths;
- **ownership clarity:** whether one team can make decisions;
- **API surface:** how much must remain compatible;
- **test isolation:** whether failures can be localized.

A **Core** module with massive fan-in and broad APIs is not a foundation. It is a congestion point.

Prefer concrete first, seam second, abstraction third

A practical progression is:

1. implement the simplest correct concrete behavior;
2. observe where change, testing, or ownership pressure appears;
3. leave a seam at the pressure point;
4. introduce an abstraction when at least two credible demands exist;
5. measure whether the boundary reduces change radius;
6. collapse it when the pressure disappears.

This is not an excuse for careless code. It is a refusal to predict an entire architecture from one requirement.

Architecture archaeology before cleanup

Inherited abstractions often look irrational because their original pressure vanished. Before deleting them, investigate commit history, incidents, platform limitations, former team boundaries, and old backend contracts. The strange compatibility mapper may be the only protection for a still-active client. The singleton may be compensating for an SDK that cannot be initialized twice. Archaeology prevents “simplification” from reintroducing a solved failure.

TRADE-OFFS

Delayed abstraction may require refactoring later. Premature abstraction imposes cost immediately and can make the wrong model durable. Concrete code is easier to read locally; explicit boundaries are easier to govern across teams. The choice should follow volatility and consequence.

FAILURE MODES

Common failures include protocol-per-type design, generic repositories with domain-free CRUD, universal base classes, utilities that become undeclared platforms, dependency injection containers used as service locators, and modules split so finely that every feature change touches a dozen packages.

WHEN NOT TO ABSTRACT

Keep code direct when behavior is local, low consequence, and unlikely to gain independent ownership. A tiny settings toggle does not need a use case, repository, mapper, and domain entity merely because checkout has them.

MIGRATION PATH

Inventory abstractions by cost and evidence. Mark each as protective, enabling, speculative, or historical. Remove speculative wrappers first. For overloaded shared modules, extract stable high-cohesion capabilities and move feature-specific behavior back to its owners. Track build and change-coupling effects so “cleanup” has proof.

ARCHITECTURE REVIEW QUESTIONS

- What exact pressure earns this abstraction?
- How many concepts and mappings does it add?
- Is the knowledge truly shared or merely similar?
- Does the boundary align with authority and ownership?
- How will we know it reduced change cost?
- What is the collapse or replacement path?

STAFF ENGINEER PERSPECTIVE

The mature move is not always another layer. It is protecting the team’s abstraction budget and making simplification an architectural outcome rather than an admission of failure.

5. Decisions, ADRs, and Revisit Triggers

Architecture is a sequence of decisions made under incomplete information. The quality of the system depends less on pretending uncertainty away than on recording assumptions, alternatives, and the conditions that should change the answer.

An Architecture Decision Record is useful when it preserves reasoning. It is useless when it merely announces an outcome.

The anatomy of a decision

A strong ADR contains:

- **context:** the pressure and constraints, with dates and evidence;
- **decision:** the chosen direction in operational language;
- **alternatives:** credible options, not straw men;
- **trade-offs:** what is gained, lost, and deferred;
- **scope:** where the decision applies and where it does not;
- **consequences:** technical, product, organizational, and operational;
- **rollout:** coexistence, migration, rollback, and observability;
- **owner:** who maintains and reviews it;
- **revisit triggers:** facts that invalidate assumptions;
- **status:** proposed, accepted, superseded, rejected, or retired.

Consider Atlas ADR-014: “Persist booking intent before UI acknowledgement.” The context is not “we prefer offline-first.” It is that response loss after server commit caused customers to retry, and support could not distinguish failure from unknown outcome. Alternatives include synchronous-only confirmation, backend lookup without a local operation, and a durable operation ledger. The decision introduces local persistence, idempotency keys, reconciliation, and an explicit `unknown` state. The revisit trigger might be a future backend protocol that provides a transaction receipt before side effects, but not “when the team feels like it.”

Decide at the last responsible moment

Some decisions are cheap to reverse: a local naming convention, an internal implementation, a small UI state holder. Others become expensive quickly: persisted schema, public SDK, cross-team module API, analytics taxonomy, remote experience grammar, model-download format.

Classify decisions by reversibility and blast radius:

CLASS	EXAMPLE	DISCIPLINE
Local/reversible	view-local state shape	code review, no ADR
Cross-feature	navigation contract	short ADR, owner
Persisted/versioned	database or operation schema	migration plan + fixtures
Cross-team/platform	shared SDK or module policy	RFC + adoption plan
External/irreversible	public API, privacy commitment	broad review + exit strategy

Documenting everything creates paperwork. Documenting nothing creates folklore. The boundary is consequence.

Rejected alternatives are future assets

A decision without alternatives loses the most valuable information: why the obvious option was not chosen. Future engineers then repeat the debate or assume negligence.

For a cross-platform proposal, record whether native, KMP, React Native, Flutter, web content, and server-driven UI were considered. Include the team's skills, feature profile, hardware needs, delivery coupling, and escape-hatch costs. When those facts change, the old analysis becomes a starting point rather than a verdict.

Revisit triggers turn documents into controls

Time alone is a weak trigger. Better triggers observe assumptions:

- build p75 exceeds the agreed feedback budget for four weeks;
- three teams need the same domain capability;

- offline completion becomes a contractual requirement;
- unsupported client share falls below a threshold;
- a feature flag reaches its removal date;
- model quality falls below a cohort-specific evaluation floor;
- a platform API removes the constraint that justified a bridge;
- an incident shows the current authority policy is ambiguous.

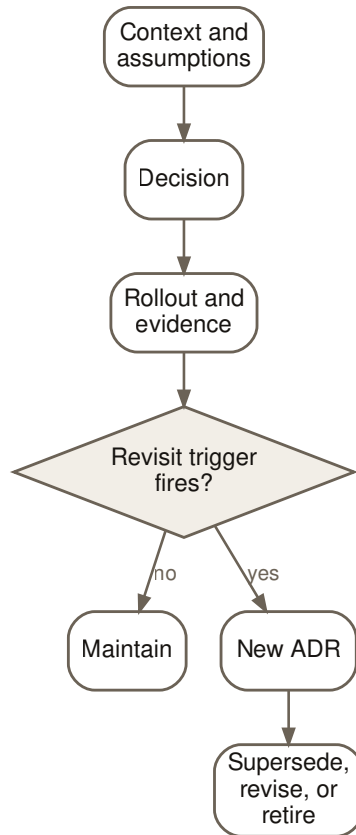


Figure 5.1 Revisit triggers turn documents into controls

A new ADR should supersede the old one rather than rewrite history. The sequence teaches architectural evolution.

Decision journals and architecture maps

ADRs answer “why this choice?” A decision journal tracks unresolved uncertainties and experiments. An architecture map shows current boundaries, authority, data movement, owners, and release controls.

For Atlas sync, the map includes local durable state, operation ledger, scheduler, API, idempotency ledger, authoritative booking record, event stream, and reconciliation endpoint. The journal may track uncertainty about average offline duration and conflict frequency. The ADR records the chosen operation protocol. Combining them into one giant “architecture document” makes all three harder to maintain.

The smallest useful approval process

A healthy review asks for consequential decisions before implementation becomes expensive. It does not require a committee to approve every pattern. Use tiers:

- local decisions: owning team;
- cross-feature decisions: affected owners;
- platform contracts: platform council or named accountable reviewers;
- security, privacy, and release-critical decisions: specialist review;
- exceptions: explicit owner and expiration.

The author remains responsible for synthesis. Consensus is not a substitute for a decision.

TRADE-OFFS

Records create continuity and can become stale. Reviews improve cross-team alignment and can become a queue. Revisit triggers make change explicit and require reliable measurements. The remedy is light artifacts, clear scope, and ownership, not abandoning documentation.

FAILURE MODES

Weak ADRs are written after the decision to create legitimacy, omit rejected alternatives, use vague consequences, have no owner, or never retire. Another failure is “architecture by precedent”: an old ADR for a large feature is copied into every new feature without checking its assumptions.

MIGRATION PATH

Start with the next consequential decision, not a historical documentation campaign. Create an index, a compact template, and statuses. As incidents or migrations touch older areas, record the recovered reasoning and mark uncertainty plainly. Add automated checks only for metadata that matters, such as owner, date, and revisit trigger.

ARCHITECTURE REVIEW QUESTIONS

- What fact makes a decision necessary now?
- Which option is cheapest to reverse?
- Which assumptions are uncertain?
- How will old and new behavior coexist?
- What evidence decides whether the choice works?
- Which event should reopen the decision?

STAFF ENGINEER PERSPECTIVE

The best ADR is not the longest. It lets a future engineer reconstruct the decision without reconstructing the crisis. Part of Staff work is protecting that context and creating a path for changing course without blaming the people who acted under earlier constraints.

6. Architecture Across Product Stages

The architecture appropriate for five engineers can be actively harmful for five hundred. Product stage changes the dominant risks, and good architecture changes with it.

Stage one: prove value

Atlas begins with two engineers and uncertain demand. The dominant risk is building the wrong product. The reasonable design is one application target, direct platform UI state, a small typed API client, and local persistence only for essential continuity. Boundaries protect high-consequence external systems (identity and payment), but there is no internal platform team, no universal domain layer, and no module for every screen.

The architectural objective is learning speed with basic correctness. Instrument the funnel, keep migrations simple, and avoid decisions that make replacement expensive. A startup does not need a miniature version of a large-company architecture.

Stage two: stabilize the product

At product-market fit, failures and coordination become more expensive. Atlas gains bookings, messages, saved itineraries, and host tools. The team introduces feature ownership, an explicit data-authority policy, durable operations for critical mutations, dependency composition, and tests around business state.

This is where MVVM or UDF may become useful, not because the company crossed a mystical size, but because state transitions and effects are now hard to reason about. Feature folders may become feature modules where build or ownership evidence supports the split. Release flags become managed assets rather than scattered booleans.

Stage three: scale teams

With several teams, the dominant risk becomes coordination and change collision. Boundaries need owners and enforced dependency direction. Shared design, networking, storage, identity, analytics, and release-control capabilities may become platform products. The build graph and local feedback loop become architecture concerns.

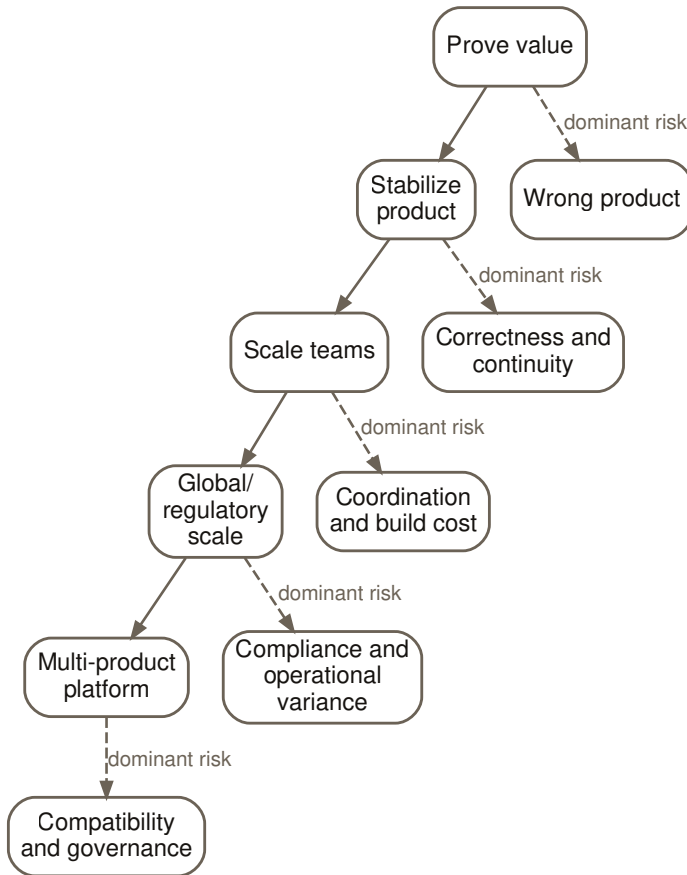


Figure 6.1 Product stages and their dominant risks

Modularity at this stage should follow domains and ownership. A module with no clear owner is an organizational ambiguity preserved in code. A platform team should provide a paved road and observable leverage, not approval theater.

Stage four: global and regulated scale

A product at this stage enters markets with different payment methods, data residency, accessibility obligations, identity rules, and network characteristics. The architecture gains policy boundaries, regional configuration, privacy inventories, contract versioning, operational runbooks, and segment-specific reliability evidence.

Global scale is not just more traffic. It is more variance. Time zones, locale, calendars, currencies, right-to-left layout, device classes, carriers, and legal rules expose assumptions that were invisible in one market. Architecture must make policy explicit without fragmenting the product into ungovernable forks.

Stage five: ecosystem and platform

A mature organization may support customer apps, host apps, internal operations, SDKs, widgets, wearables, and partner integrations. Shared capabilities now have multiple consumers and release clocks. Compatibility governance, deprecation, semantic versioning, contract tests, and platform adoption metrics become central.

At this scale, the mobile engineering system includes scaffolding, dependency catalogs, module linting, observability libraries, release automation, model distribution, experimentation, and incident tooling. The platform succeeds only when product teams ship more safely and cheaply, not when it owns the most code.

Introduce and remove by pressure

A stage model is not a maturity ladder. Every practice should have an introduction trigger and a removal condition.

PRACTICE	INTRODUCE WHEN	DELAY OR REMOVE WHEN
Feature modules	ownership, build, or change coupling demands isolation	configuration cost exceeds isolation value
Central design system	repeated UI semantics need consistency	it blocks necessary product variation
Sync engine	durable offline mutations or unknown outcomes exist	the product can be honest with online-only behavior

PRACTICE	INTRODUCE WHEN	DELAY OR REMOVE WHEN
Platform team	repeated cross-team capability work dominates	adoption is low and local ownership is cheaper
Cross-platform core	stable shared domain behavior has real consumers	platform divergence and bridge cost dominate
Architecture council	cross-domain decisions need accountable arbitration	it becomes an approval queue for local choices

Architecture evolution is a product roadmap

Treat evolution as staged delivery. A module extraction needs compatibility adapters and consumer sequencing. A new state architecture needs coexistence with old navigation. A database replacement needs dual reads, shadow verification, and rollback. A cross-platform initiative needs a bounded pilot and exit criteria.

The roadmap should connect each architectural investment to a product or operational outcome. “Modernize the app” is weak. “Reduce booking-change lead time from six weeks to two while maintaining the duplicate-reservation invariant” is actionable.

TRADE-OFFS

Early simplicity can create later migration work; early infrastructure can consume the runway before it creates value. Centralization reduces duplication and can reduce autonomy. Standardization improves mobility and can suppress context.

FAILURE MODES

Teams cargo-cult big-tech structures, split modules before ownership is stable, create platform teams without product discipline, or retain startup shortcuts after they endanger revenue and trust. A subtler failure is assuming growth is monotonic. Products contract, merge, and change direction; architecture must support consolidation and deletion too.

MIGRATION PATH

Map the current stage by dominant risk, not employee count. Identify one pressure that the existing design no longer absorbs. Define the smallest

capability or boundary that addresses it, create a measurable outcome, and plan coexistence. Reassess quarterly, because the dominant risk can move faster than the codebase.

ARCHITECTURE REVIEW QUESTIONS

- What is the product's dominant risk now?
- Which practice has the pressure to justify its cost?
- Does the design assume a team size or release model we do not have?
- How does the architecture evolve at the next plausible stage?
- Which current machinery should be removed if the product contracts?
- How is architectural investment tied to a product or reliability outcome?

STAFF ENGINEER PERSPECTIVE

Architecture leadership is timing. Introduce a boundary before coordination failure becomes normal, but not so early that the organization spends its best attention servicing a future that may never arrive. The Staff role is to make that timing explicit and reversible.

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.

Where to Go from Here

The method pays off only when it meets a real decision. Start with your own product this week: pick one critical journey, fill in the decision canvas, and write down a revisit trigger for a decision that matters.

Keep building with the Atlas system

The Pro edition ships with the Atlas reference system: runnable iOS, Android, Kotlin Multiplatform, and backend code with contract schemas. The Extended edition adds React Native, Flutter, and the sync simulator that injects duplicate events, reordered writes, process death, and model failure. Read a chapter, open the matching project, run the tests, then break an assumption and watch the architecture hold or fail.

Also by Mike Salari

If this book is about the decisions, these are about the career that lets you make them and be heard.

Leverage

THE NEGOTIATION OPERATING SYSTEM FOR ENGINEERS

Turn a competing offer, a promotion case, or a raise conversation into a system instead of a gamble.

The Senior Signal

What Android interview loops actually score, from the hiring side of the table, with the traps and the recovery lines.

iOS Interview Blueprint

The same hiring-side treatment for iOS: the questions as they are really asked, and the answers that separate senior from mid.

Ticket Taker to Outcome Owner

How to stop being handed tickets and start owning outcomes, the shift that defines the move to senior and staff.

Leadership Blueprint

OUTCOMES WITHOUT AUTHORITY

Leading the decision when you do not control the budget, the roadmap, or the people, which is most of the time.

Build With Your Brain On

Engineering judgment in the age of AI-accelerated implementation, where writing the code stopped being the hard part.

ALL TITLES ARE AVAILABLE AT [SALARI.DEV/BOOKS](https://salari.dev/books)