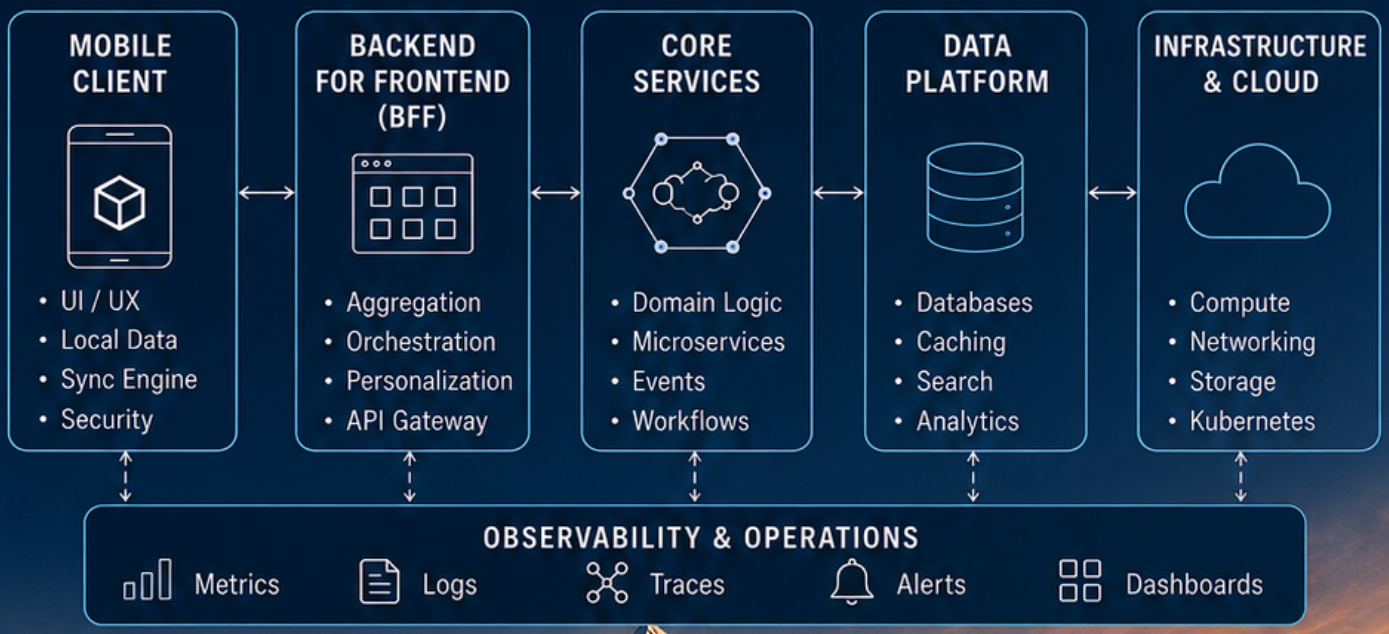
ALTITUDE

A FIELD GUIDE TO STAFF-LEVEL MOBILE SYSTEMS ENGINEERING

Think in Systems. Design for Reality. Build What Scales.



REAL-WORLD ARCHITECTURES FROM UBER, WHATSAPP & REVOLUT

FOR ENGINEERS WHO BUILD SYSTEMS THAT MILLIONS OF PEOPLE RELY ON, EVERY DAY.

MIKE SALARI

FIRST EDITION
2026

Altitude

A field guide to staff-level mobile systems engineering

First edition, 2026

The case studies in this book are representative architectures, inspired by public engineering talks, papers, open-source projects, and platform documentation. They are teaching models, not claims about any company's internal systems.

salari.dev mike@salari.dev

Why read this book

Move beyond features. Think in systems.

You can already build anything they ask for. This book teaches the half that gets you to staff: how the mobile client fits into the whole system, and how that system behaves, scales, and fails in production.

What you will learn

- Design mobile systems that scale across teams and survive production
- Think like a staff engineer: trade-offs, failure modes, and real judgment
- Understand distributed systems from the client's point of view
- Master backend-for-frontend (BFF) architecture
- Design secure, attested, and observable mobile platforms
- Learn from complete end-to-end walkthroughs of Uber, WhatsApp, and Revolut style systems

Featured architecture walkthroughs

- Uber-inspired ride-hailing platform
- WhatsApp-inspired messaging platform
- Revolut-inspired financial platform

The full end-to-end walkthroughs are in the complete edition. A preview of the ride-hailing walkthrough follows.

Preface

This book started with a message from a stranger.

A senior iOS engineer, preparing for interviews, sent me a note. He had spent years getting good at mobile engineering, architecture, performance, concurrency, testing, and now he kept running into the same wall. The job descriptions and the interview panels wanted more. System design. Backend for frontend. Authentication and threat modeling. Messaging systems. Kubernetes. He asked a simple question: does this actually matter for senior and staff roles, and if so, what should I learn first?

I wrote him a long answer. The short version was this. For senior roles, your mobile depth is the thing. No amount of Kubernetes trivia compensates for a weak foundation. But at staff level the expectation changes. You are no longer responsible only for the app. You are responsible for how the client fits into the entire system, and the engineers who make that jump are the ones who can connect product, mobile architecture, backend, security, and reliability into one coherent answer.

He read it, thanked me, and asked for the full course.

There was no full course. The material he wanted was scattered across distributed-systems textbooks, a hundred engineering blog posts, RFCs, conference talks, and the kind of knowledge that usually only transfers by sitting next to someone who has already made the mistakes. So I built it. This is that book.

Who this is for

You are a strong senior mobile engineer, iOS or Android. You can ship features, you own architecture on your team, and you have started to feel the ceiling. You want the systems judgment that lets you lead a design that spans teams, predict where it breaks, and hold your own with backend, platform, and security engineers without bluffing.

You do not want to become a backend engineer or a DevOps engineer. Good. This book does not try to make you one. It teaches you what those systems are, why they exist, how they fail, and how your app fits into them, so that you can make the call and defend it.

The one promise

By the last page, you can reason about the entire system your app depends on, make and defend staff-level architecture trade-offs, and connect product, mobile, backend, security, and reliability into one answer. That is the whole job at staff level, and it is learnable.

How to use this book

Read it in order the first time. It is built as a curriculum, not a reference. Part 1 rewires how you see the client. Part 2 teaches you to design the mobile system. Part 3 is the contract with the backend. Part 4 is data, sync, and resilience, the hardest and most valuable material. Part 5 is real time, security, and observability. Part 6 zooms out to the cloud and to the role itself.

Every chapter opens with what you will be able to do by the end, teaches from concrete examples before abstractions, ends with the judgment calls a staff engineer actually makes, a set of interview questions with notes on what a strong answer covers, and the sources to go deeper. After the first read, the chapters stand alone as references.

A note on sources

Nothing in this book is invented. Every case study, number, and claim traces back to primary engineering sources: the teams at Meta, Uber, Airbnb, Netflix, Stripe, Shopify, Slack, Cloudflare, Linear, Figma, Amazon, Google, Apple, and others, plus the standards bodies and the small canon of books that define this field. The full ranked bibliography, the reading roadmaps by level, and the research behind every chapter live in the companion research foundation. Where the industry has no consensus, this book says so rather than pretending the question is settled. That honesty is the point. Staff engineers are paid for judgment under uncertainty, not for reciting settled facts.

Let us begin where every staff engineer begins, with the moment you stop thinking like the owner of a feature and start thinking like the owner of a system.

This is a sample

You are reading a free sample of Altitude, a field guide to staff-level mobile systems engineering, first edition.

This sample is built to let you judge the book before you read the whole thing. It includes the preface, the complete table of contents, the first two chapters in full, and the opening of one case study. Chapter 2 in particular carries diagrams, a worked example, and exercises with solutions, so you can see the depth and the format, not just the marketing.

The full edition runs to 28 chapters across seven parts, roughly 165,000 words. It adds 28 architecture diagrams, three full-page end-to-end case-study blueprints (a ride-hailing platform, a global messaging platform, and a high-assurance financial platform), worked exercises throughout, a 74-term glossary, and a fully cited research foundation behind every claim.

The complete table of contents is on the next page, so you can see exactly what the rest of the book covers. When you are ready for the whole thing, the last page tells you how to get it.

Contents of the full edition

The full book contains all of the following. This sample includes chapters 1 and 2 in full and the opening of chapter 26.

Chapter 1: From feature owner to systems thinker

Two engineers interview for the same staff role. The first can recite a Kubernetes deployment manifest from memory, name every field in a horizontal pod autoscaler, and draw the control loop on a whiteboard without hesitating. The second cannot, but when you ask why a mobile app would put a backend-for-frontend between the phone and the services, she talks for ten minutes: about chatty round trips on a subway connection, about a payload shaped for a watch versus a tablet, about who gets paged when the aggregation layer falls over, about the release you cannot take back once it is on a million phones. One of them gets the offer. It is not the one who memorized the manifest.

That is the whole shift in a sentence. The engineer who knows the technologies loses to the engineer who knows why each one exists. At senior level you are rewarded for being the person who can build the thing. At staff level you are rewarded for being the person who can see the whole thing, name where it breaks, and connect product, mobile, backend, security, and the business into one answer that holds up in a room full of people who each own only one slice. The technologies are still on the table. They are just no longer the point.

This chapter is about making that turn deliberately, because most people make it by accident or not at all. You will meet engineers with a decade of experience who never made it, who stayed brilliant feature owners forever, blocked not by skill but by a missing habit of mind. The habit has a name. It is systems thinking, and it is learnable.

By the end of this chapter you can:

- Name the concrete difference in what a company expects from you at senior versus staff, in terms of the decisions you are trusted to make.
- Explain systems thinking as a working skill, not a slogan, and apply it to a mobile design problem.
- Walk the staff engineer's reasoning order (mobile system design, then the contract with the backend, then auth and security, then observability, then caching and sync, then event-driven, then cloud) and say why each layer exists rather than reciting the tools that implement it.
- Spot the trap of collecting technologies instead of understanding boundaries, and avoid it in your own learning.

What actually changes between senior and staff

A senior mobile engineer optimizes a screen. A staff engineer designs the constraints that let two hundred other people optimize their screens without colliding. That line, drawn straight from the

case studies this book rests on, is the cleanest summary of the gap, and it is worth sitting with, because it is not about working harder or knowing more APIs. It is about a change in the unit of work. Your unit of work used to be a feature. Now it is the system the features live in.

The mobile system design research foundation frames the difference around reversibility. At staff level you are paid for the decisions that are expensive to reverse. A networking library choice is cheap; you can swap it in a sprint. A module boundary that fifty teams now depend on is not. Mobile makes irreversibility worse than backend for three structural reasons the foundation lays out, and you should be able to recite them in your sleep because they shape everything that follows. First, you ship a monolith to the device whether you like it or not: even a beautifully modular codebase links into one process and one binary, so a memory leak or a startup regression in any module is everyone's problem. Second, you cannot roll forward instantly; users sit on old versions for months, and a bad release propagates at the speed of store review plus user update behavior, not at the speed of a deploy. Third, your build and CI become a shared resource a growing team will saturate, so build performance is an architecture concern, not a tooling afterthought.

Read those again as a job description, not a list of facts. A senior engineer is judged on the screen they shipped this sprint. A staff engineer is judged on whether the boundary they drew two years ago is still letting teams move, or has quietly become the thing everyone routes around. The senior bug is a crash in your feature. The staff bug is a startup-time regression that no single commit caused, where six teams each added a little synchronous work to app launch and no code review caught the sum, because the launch path is a shared resource with no default owner. That failure mode, which the foundation calls startup creep, is invisible to a feature owner and obvious to a systems thinker, and the difference between the two is the difference this chapter is teaching.

Here is the part nobody says out loud. The senior-to-staff jump is not a promotion you earn by doing your current job better. Doing your current job better makes you a stronger senior. The staff job is a different job, and the companies that promote well know it. When Tanya Reilly maps the staff role in The Staff Engineer's Path, and Will Larson does the same in Staff Engineer, both books that anchor the principal reading roadmap in this book's research foundation, the throughline is that staff work is leverage, not output. You stop being measured by what you build and start being measured by what you make possible for everyone else.

Systems thinking, made concrete

Systems thinking sounds like a poster on a wall. Let me make it a skill you can watch someone use.

Product wants to A/B test a new checkout flow to five percent of users and kill it instantly if conversion drops. A feature owner hears that and starts building the checkout. A systems thinker hears it and asks a different first question: what is the smallest, most reversible way to put this in front of users and pull it back without shipping a new binary? The answer, drawn from Uber's architecture work in the foundation, is that you map the request onto a core-versus-optional split with a master feature flag, not a code branch. Core code is everything the business cannot survive losing, the sign-up, the trip, the payment, gated behind a stricter review. Optional code is everything you can turn off without stopping the business, gated behind flags you can kill. The new checkout is optional until proven. You instrument the conversion metric that triggers the kill before you write the feature, not after. And because you cannot hotfix a phone, the kill has to be a server-side flag flip, because a code rollback would take a store review and days of user updates, by which point the damage is done.

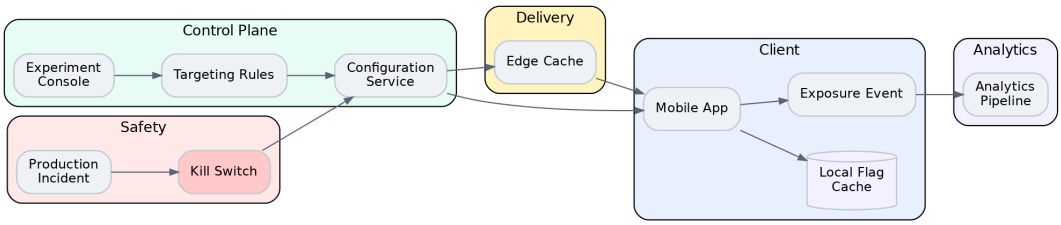


Figure 1: Feature flags, rollout control, kill switches, and observability

Watch what just happened. The systems thinker never touched the checkout UI in that reasoning. She reasoned about reversibility, ownership, the release constraint, and the metric, and the UI fell out at the end as a detail. That is systems thinking: seeing the feature as a node in a system of constraints, and solving the constraints first. Donella Meadows wrote the canonical book on this, Thinking in Systems, which sits in this foundation's principal roadmap as a meta-skill, and her core move is exactly this: look at the structure that produces the behavior, not the behavior itself. The massive view controller is not a discipline problem, it is a structure problem, a pattern with no home for business logic except the controller. Startup creep is not a laziness problem, it is a structure problem, a shared resource with no owner. Fix the structure and the behavior follows.

The other half of systems thinking is Conway pressure, and it is the half engineers resist longest because it feels like politics rather than engineering. The foundation states it flatly: your architecture and your org chart converge whether you plan it or not, so design them together. Spotify's famous squads-and-tribes model, the one half the industry tried to copy, is the cautionary tale here. The foundation is blunt that companies cargo-culted the labels and got none of the autonomy, alignment, or platform investment that made the model work, and that Spotify themselves treated it as a snapshot, not a doctrine. The lesson is not "do the Spotify model" or "do not." It is that the org model is not the platform; the labels are cheap and the platform is expensive, and only the platform creates real autonomy. A systems thinker reads an org chart as part of the architecture, because Conway's law guarantees it is.

So when someone asks you to "think more strategically," this is the concrete thing they are asking for. Not bigger words. The habit of asking, before you build: what produces this behavior, what is expensive to reverse here, who owns the seams, and what does the org chart already commit us to. That is the whole discipline. Everything else in this book is practice at applying it to specific layers of the stack.

The order a staff engineer reasons in, and why each layer earns its place

There is a priority order a staff mobile engineer reasons through, and it is the spine of this book: mobile system design first, then the contract with the backend (API and BFF), then auth and security, then observability, then caching and sync, then event-driven systems, then cloud and Kubernetes. People treat this as a syllabus to memorize. It is not. It is a sequence of why-questions, and the value is in being able to answer each "why does this layer exist" without naming a single product. Let me walk it that way, because that is the walk an interviewer is listening for.

Mobile system design comes first because it is the layer where the expensive, irreversible decisions live: the module boundaries fifty teams depend on, the core-versus-optional split, the native-versus-cross-platform position, the rendering choice between native and server-driven UI. The foundation's whole first section is one tension, how do you keep a core experience reliable to four nines while letting a hundred teams experiment recklessly on top of it, and every company answered it differently. Uber answered with Riblets and a hard core-optional split. Airbnb answered by moving rendering decisions to the server with their Ghost Platform. Meta's FBiOS team answered by replacing runtime wiring with a build-time plugin graph, converting whole categories of runtime failures into build-time warnings, which at billions of users is worth more than any pattern elegance because a runtime failure ships to billions and a build error stops one engineer. You do not memorize Riblets. You learn that this layer is where you decide how risk is distributed, and you reason from there.

The contract with the backend comes next because the mobile client is useless alone; it lives or dies on the round trips it makes. This is where the backend-for-frontend earns its existence, and the BFF is the single best test of whether someone reasons or memorizes. A memorizer says "a BFF is an API gateway for mobile." A reasoner says a BFF exists because a phone on a flaky network cannot afford the chattiness of calling six microservices to render one screen, because the payload a phone needs is shaped differently from the payload a web app needs, because someone has to own the aggregation and translation that would otherwise smear across every client, and because that ownership boundary is exactly where mobile and backend teams negotiate. Sam Newman's BFF writing and Phil Calçado's pattern essay, both top-tier sources in this foundation, exist to answer the why, not the what. The hook this chapter opened with is real: the engineer who can explain why a BFF exists beats the one who memorized the manifest, because the BFF is a boundary decision and boundary decisions are the job.

Auth and security come third because the client is the most hostile node in the system, a phone you do not control running a version you cannot patch, and everything you trust it with is a thing an attacker can reach. The foundation's auth canon, RFC 8252 for OAuth on native apps, PKCE, refresh-token rotation, passkeys, exists because mobile auth is genuinely different from web auth, and the reason it sits this high in the order is that getting it wrong is both catastrophic and irreversible in the worst way, since the credential is already on devices you cannot recall.

Observability comes fourth, and its placement is the most counterintuitive until you internalize the refrain you will hear all through this book: you cannot hotfix a phone. The foundation's whole observability framing is "telemetry you cannot hotfix." On the backend, if you are blind to a problem you add logging and redeploy in minutes. On mobile, the instrumentation you forgot to ship is instrumentation you do not have for months, until the next release reaches users. So observability is not a thing you bolt on after launch; it is a thing you design into the binary before launch, because the binary is the only chance you get. That inversion, instrument before you ship because you cannot instrument after, is why this layer sits above the data layers rather than below them.

Caching and sync come fifth because once the client is reliable and observable, the next hard problem is that it holds a copy of data that drifts from the server's copy, and reconciling those two copies without losing the user's work or showing them a lie is, in the foundation's words, the whole game. Event-driven systems come sixth because real-time delivery and the catch-up-after-offline problem ride on top of the sync foundation. And cloud and Kubernetes come last, deliberately, because for a mobile architect they are the mental model, not the manifests. You need to know what an autoscaler does to your tail latency and what a regional failover does to your client's cursor. You do not need to write the YAML. The engineer in the cold open had the order exactly

backwards, and that is why she lost.

The point of the order is not the order. The point is that each layer is a why, and a staff engineer who can answer all seven whys can reason about a system they have never seen, while an engineer who collected seven technologies can only recognize the ones they have met before.

The trap: collecting technologies instead of understanding boundaries

The most common way to stall on the way to staff is to treat the role as a checklist of technologies to acquire. Learn Kubernetes. Learn GraphQL. Learn Kafka. Learn OAuth. The list feels like progress because each item is concrete and you can check it off. It is a trap, and the foundation's case studies are full of engineers and whole companies that fell into it.

The cleanest example is the cross-platform debate, because the foundation gives you both sides by name and they refuse to cancel out. Airbnb made a public, well-argued bet on React Native in 2016 and publicly reversed it in 2018, and the detail that keeps the story honest is the survey: sixty-three percent of their engineers would choose React Native again, and they sunset it anyway. Shopify made the same bet in 2019 and, by their 2025 retrospective, had migrated every app to it with sub-500ms screen loads and crash-free rates above 99.9 percent, while stating flatly that "100% React Native should be an anti-goal." A technology collector reads those two stories and demands a verdict: is React Native good or bad? There is no verdict. The foundation is explicit that the two stories bound the decision rather than settling it, and the variables that move the answer are how large and native-specific your surface is, how web-fluent your team is, and whether you treat full cross-platform as a goal or keep native escape hatches for hardware and background work. A systems thinker does not collect "React Native" as a skill. They learn to place a surface on the spectrum from maximum control to maximum shared velocity and defend the placement for their org. The skill is the boundary judgment. The framework is just one of the things you reason about.

The same trap hides in every layer. The foundation warns that adopting the Spotify org model as a literal template, rather than its principles, is widely regarded as a mistake by the people who watched it spread. It warns that "one general-purpose mobile architecture document and hope teams comply" is outdated, because governance now means enforced boundaries, build-system-checked dependencies and lint rules, not a wiki page. It warns, in the distributed systems section, against "just retry until it works," because at fleet scale unbounded retries are a leading cause of cascading failure, which is a lesson about understanding the system's dynamics, not about knowing a retry library. In every case the collectable thing, the framework, the org chart, the retry call, is the surface, and the boundary judgment underneath is the staff skill.

So how do you actually learn, if not by collecting? You learn the way this book is built. For every technology, you ask the why-question until you can answer it without the technology's name. Why does a BFF exist: to own aggregation and translation at the mobile-backend seam so chattiness and payload-shaping do not smear across clients. Why does a feature flag exist: to make a risky change reversible without a release on a device you cannot hotfix. Why does observability ship in the binary: because you cannot add it later. When you can answer the why, you can evaluate any tool that claims to solve it, and you can tell when a new tool is genuinely better versus merely new. That is the difference between an engineer who is current and an engineer who is durable, and staff engineers are durable.

In practice: a cross-team design review

Watch a staff engineer run a real one, because the abstract turns concrete fast.

A second app is coming. The company has a successful consumer app, and now there is a driver app (or a merchant app, the shape is the same), and leadership wants it to reuse sixty percent of the consumer app's logic. The room has the consumer mobile lead, two backend engineers, a security partner, and a product manager. Everyone wants something different. The feature owner in the room wants to start coding the new screens. The staff engineer slows the room down and reasons through the layers in order, out loud, because the reasoning is the deliverable.

She starts with system design, the expensive-to-reverse layer. The real decision is not "how do we share code," it is "what becomes a shared platform module versus what stays app-specific," and that boundary is the thing the next two years route through. She maps it onto the foundation's platform-versus-autonomy tension: identity, networking, navigation, analytics, and the release train are shared rails the new app sits on, and she names them as a platform with internal customers, an SLA, and a roadmap, because the foundation is clear that if you do not treat the platform as a product, every team builds a worse version of it privately. She does not let the room turn this into a vague "let's share what we can," because a boundary you cannot mechanically verify is a boundary that erodes.

Then the backend contract. The driver app's screens need different data shaped differently from the rider's, so the question of whether they share a BFF or get their own is live, and she frames it as an ownership question, not a technology one: who gets paged when the aggregation layer for the driver app falls over, and is that the same team that owns the rider BFF. The org answer drives the technical answer, Conway pressure made visible in a single meeting.

Security comes next, and she pulls the security partner in early rather than at the end, because a driver app touches earnings and payouts, which moves more of its surface into the core, stricter-review half of the split. Then observability: she asks what the new app must instrument in its very first binary, because whatever they forget to ship is blind for months. Only after all of that does she let the room talk about screens, and by then the screens are easy, because every hard decision has been named, traded off, and assigned an owner.

The feature owner shipped a screen. The staff engineer shipped a system the screens can live in. Same meeting, two different jobs, and the second one is the one that scales to a hundred teams.

What a staff engineer decides

- The unit of work is the system, not the feature. Spend your judgment on the decisions that are expensive to reverse, the module boundaries and the core-optional split, and leave the cheap, swappable choices to the teams.
- Reason in why-questions, not technology names. Before you adopt anything, be able to say why that layer exists without naming the product that implements it. If you cannot, you are collecting, not understanding.
- Design the org chart and the architecture together, on purpose, because Conway's law will do it for you otherwise, and it will do it worse.
- Treat reversibility as a first-class design property. On mobile, a feature flag is worth more than elegance, because you cannot hotfix a phone and a flag is the only fast way back.
- Name the platform as a product with internal customers and an owner, or watch every team

rebuild a worse version of it privately.
- When the field disagrees with itself, as it does on cross-platform and on who owns the BFF, your job is to hold both strongest cases and pick for your context, not to find the universal verdict. There usually is not one.

Common interview questions

- What changes between a senior and a staff mobile engineer? A strong answer names the shift in the unit of work from feature to system, points at reversibility as the thing staff judgment is spent on, and gives a concrete mobile example such as a startup-time budget no single commit owns.
- Why does a backend-for-frontend exist? A strong answer reasons from chattiness on a flaky network, client-specific payload shaping, and the ownership boundary at the mobile-backend seam, rather than reciting "it is an API gateway."
- A product team wants to ship a risky checkout change to five percent of users and kill it instantly. How do you design that? A strong answer maps it onto a core-optional split with a master flag rather than a code branch, instruments the kill metric first, and notes that the kill must be server-side because you cannot hotfix a phone.
- How do you decide native versus cross-platform for a new surface? A strong answer places the surface on the control-to-shared-velocity spectrum, weighs native surface size and team web-fluency, names the bridge and upgrade tax, and cites both the Airbnb exit and the Shopify commitment as bounded outcomes rather than a verdict.
- A second app needs to reuse most of the first app's logic. How do you approach it? A strong answer separates shared platform modules from app-specific code, treats the platform as a product with an owner and an SLA, and uses Conway pressure to drive the BFF-ownership question.
- Leadership wants to "do the Spotify model." What do you tell them? A strong answer translates the request into platform-versus-org terms, adopts the principles of autonomy-with-alignment rather than the labels, and pushes back on copying labels without the platform investment that makes them work.

Key takeaways

- At senior you are the strongest engineer in the room; at staff you own how the client interacts with the entire ecosystem.
- Systems thinking is a habit, not a slogan: look at the structure that produces the behavior, solve for reversibility, and read the org chart as part of the architecture.
- The seven-layer reasoning order is a sequence of why-questions, not a list of technologies to acquire. Master the whys and you can reason about systems you have never seen.
- The engineer who knows why a BFF exists beats the one who memorized a Kubernetes manifest, because boundary judgment is the staff skill and tool knowledge is not.
- Collecting technologies stalls careers; understanding boundaries advances them. When the field has no consensus, hold both cases and pick for your context.
- The platform is a product. Name it, own it, or watch it get rebuilt badly in private.

Further reading

The sources behind this chapter, all from the research foundation:

- "The evolution of Facebook's iOS app architecture." Meta Engineering, 2023. https://engineering.fb.com/202 ios-app-architecture/
- "Engineering the architecture behind Uber's new rider app." Uber Engineering, 2016. https://www.uber.com/us/en/blog/new-rider-app-architecture/
- "A deep dive into Airbnb's server-driven UI system." Airbnb Engineering, 2021. https://medium.com/airbnb-engineering/a-deep-dive-into-airbnbs-server-driven-ui-system-842244c5f5
- "Sunsetting React Native." Gabriel Peal, Airbnb Engineering, 2018. https://medium.com/airbnb-engineering/sunsetting-react-native-1868ba28e30a
- "Five years of React Native at Shopify." Mustafa Ali, Shopify Engineering, 2025. https://shopify.engineering/five-years-of-react-native-at-shopify
- "Backends for frontends." Sam Newman. https://samnewman.io/patterns/architectural/bff/
- Team Topologies. Matthew Skelton and Manuel Pais, IT Revolution, 2019. https://teamtopologies.com/book
- "Scaling Agile @ Spotify with Tribes, Squads, Chapters & Guilds." Kniberg and Ivarsson, Crisp, 2012. https://blog.crisp.se/wp-content/uploads/2012/11/SpotifyScaling.pdf
- The Staff Engineer's Path. Tanya Reilly, O'Reilly. And Staff Engineer: leadership beyond the management track. Will Larson.
- Thinking in Systems. Donella Meadows.

You now have the mindset. The next move is to make the central reframe of this whole book concrete, the one everything else builds on: the mobile client is not a screen you are decorating, it is a node in a distributed system, and the most hostile node in it. That is chapter 2.

Chapter 2: The mobile client as a distributed-systems node

Here is the thing nobody tells you when you move from senior toward staff: you have been building distributed systems for years without calling them that. A phone with a local cache talking to a backend over an unreliable network is a distributed system with two nodes, no shared clock, and a link that fails constantly. Every offline-capable app is a replicated, eventually-consistent store with conflict resolution. Every retry you write participates in the same dynamics that take down datacenters. The reason your sync logic has subtle bugs is that you have been solving CAP, consensus, and consistency problems with intuition instead of theory.

This chapter asks you to do one thing, and it is the most important reframe in the book: stop seeing the client as a screen and start seeing it as a node. A screen is something you decorate. A node is something that participates, that holds state, that can be partitioned from its peers, that has to behave correctly when the rest of the system cannot reach it and when it cannot reach the rest of the system. The moment you make that switch, a whole class of bugs stops being mysterious. The spinner that never resolves, the edit that silently vanished, the outage your own app made worse, all of them are node problems, and nodes have a literature.

The client is a node, not a screen. Say it until it is reflexive, because everything that follows is a consequence of it.

By the end of this chapter you can:

- Reason about latency, network partition, and partial failure from the client's seat, instead of treating the network as a wire that occasionally hiccups.
- State the CAP theorem precisely, explain why the client almost always sits on the availability side, and describe what reconciliation it therefore owes the user.
- Name the specific ways the client is the most hostile node in the system: untrusted, unreachable, and unfixable after ship.
- Predict how your app behaves when the network splits, the backend fails over, or a million phones reconnect at once, and design for those moments before they happen.

The node nobody wants to admit they have

Picture the simplest possible "system" you ship: an app and a server. One phone, one backend, one network link between them. It does not feel like a distributed system. It feels like a client and a server, which is the oldest, most boring topology there is. That framing is exactly the problem, because the moment the phone keeps any local state, a cache, a draft, a queued action, you have two replicas of the truth, no clock you can trust to order them, and a link that fails not as an

exception but as a steady-state condition. The foundation says it without hedging: a phone with a local cache talking to a backend over an unreliable network is a distributed system with two nodes, no shared clock, and a link that fails constantly.

So why does it not feel that way? Because the demo always works. On your desk, on office wifi, the network is a wire, the latency is single-digit milliseconds, and the partition never happens. Every bug that lives in the gap between the two replicas is invisible in the place where you build. This is the deepest trap in mobile engineering, and it has a name you will hear all through this book: design for the partition, not the demo. The demo is the happy path on a perfect network. The partition is the elevator, the plane, the subway, the dead zone, the carrier hiccup, the backend failover, and your users live there a meaningful fraction of every day.

What makes this more than a slogan is that the client sits on the wrong side of every hard problem in distributed systems. The foundation is precise about this. The client is the replica with the stalest data, the node most likely to be partitioned, the one with no durable clock you can trust. A backend node lives in a datacenter with redundant fiber, synchronized clocks, and a sibling it can reach in a millisecond. The client lives in a pocket, on a battery, on a cell network it shares with a stadium full of people, holding data that might be a week old, with a clock the user can set to any value they like. Every assumption that makes backend distributed systems tractable, low latency, reliable links, trustworthy time, is false on the phone. You are operating the worst node in the system, and you are operating millions of copies of it at once.

That last fact is the one that changes your job. A backend engineer who has a capacity problem adds machines. You cannot add phones that behave. The foundation puts the consequence starkly: the client is the load, so the client owns the blast radius. Backend engineers can add capacity; you cannot add phones-that-behave. When a million of your nodes do the same wrong thing at the same instant, you are not a participant in an outage, you are the cause of one. Hold that thought; we will come back to it as the chapter's sharpest point.

Latency, partition, and partial failure from the client's seat

Three things go wrong on a network, and a feature owner treats all three as "the request failed." A systems thinker treats them as three different problems with three different responses, because conflating them is how you ship the spinner that never dies.

The first is latency. The request will succeed, but later than you hoped, and the distribution has a long tail. The foundation's resilience guidance is blunt: always set timeouts, and make them shorter than you think, because a request with no timeout is a resource leak waiting to wedge your UI, and you should pick the timeout from the latency distribution, say the p99 plus headroom, not a round number you guessed. The reason this matters more on mobile than anywhere else is that the user is watching. A backend service waiting on a slow dependency is a metric; a phone waiting on a slow dependency is a human staring at a spinner deciding whether your app is broken. Latency on the client is a UX surface, and a timeout is the difference between "this is taking a moment" and "this is hung."

Latency on a phone is also worse than the round-trip number suggests, because the cellular radio itself adds time and cost that wired clients never pay. A cellular interface runs a radio resource control state machine, and waking the radio from idle to a connected state takes time before your bytes ever move. Then, once your transfer finishes, the radio does not go straight back to sleep. It sits in a high-power state for an inactivity period, the "tail," that burns battery while

transferring nothing. Measured values from the literature are network-configured rather than fixed, on the order of several seconds and up to about fifteen seconds depending on the carrier, and that tail can account for a large fraction of an app's radio energy (ACM MobiCom 2015 RRC study, sigmobile.org). The directional lesson is exact: every separate little request pays the wake-up cost and leaves a tail behind it, so a chatty client that fires ten small calls a second apart keeps the radio pinned awake far longer than the calls themselves take. This is why batching and coalescing are not micro-optimizations on mobile but first-order design. The network is not just slow and unreliable, it is metabolically expensive, and the client pays in the user's battery for treating it like a free local function call.

The second is partition. The request will not succeed, because right now there is no path between this node and its peer. The user walked into the elevator. The plane door closed. This is not a degraded version of latency, it is a different state, and the distributed systems literature treats it as its own thing for a reason we are about to make formal with CAP. The client's job during a partition is not to fail; it is to keep working from local state and queue what it cannot send, so that the work survives until the link comes back. An app that becomes a useless rectangle the moment the network drops has mistaken a partition for an error.

The hard part of a partition is not the gap, it is the rejoin. While the link is down, the user keeps acting and the server keeps changing, and the two replicas drift. When the link returns, the client owes a reconciliation, and the shape of that reconciliation is the whole design. Trace one write across the partition and back:

Notice what the diagram makes unavoidable. The client accepts the write before it knows the server will, which means it has taken on a debt. It carries an idempotency key so the replay is safe to repeat. It waits for a server version on the way back so it can tell whether the server moved underneath it. None of that is in the feature owner's two-line version, and all of it is forced the moment you admit the partition is a normal state, not an error.

The third, and the subtlest, is partial failure: some of it worked and some of it did not, and you do not always know which. The classic case from the foundation is the offline saga. The user makes several linked changes offline, create a trip, add three stops, attach a photo, and on reconnect you replay them as an ordered sequence against the server. If step three fails because the photo upload is rejected, you do not abandon the whole batch; you run compensating actions, mark the photo as failed, keep the trip and the stops, surface a retry. That is a saga: a sequence of local transactions with compensation instead of a single distributed transaction, which is exactly how distributed systems handle multi-step operations without a global lock. The feature owner writes code that assumes the batch either all succeeds or all fails. The systems thinker knows that "partially" is the normal outcome and designs the compensation before the failure, not during the postmortem.

Partial failure is worse than it sounds because the ambiguity is fundamental, not a gap in your logging. The client sends a write and then the link dies. There are two worlds consistent with what the client observed: one where the server never received the request, and one where the server applied it and the acknowledgment died on the way back. From the phone's seat those two worlds are identical. No amount of inspecting the local state distinguishes them, because the difference lives on the other side of a partition you cannot cross. This is the two generals problem in a pocket, and it is why "did my write land" is not a question you can answer by trying harder. You answer it structurally, by making the answer not matter.

There is a fourth horseman that hides inside that ambiguity, and it is the one that does real damage: the retried write that double-applies. If the network drops after the server processed your

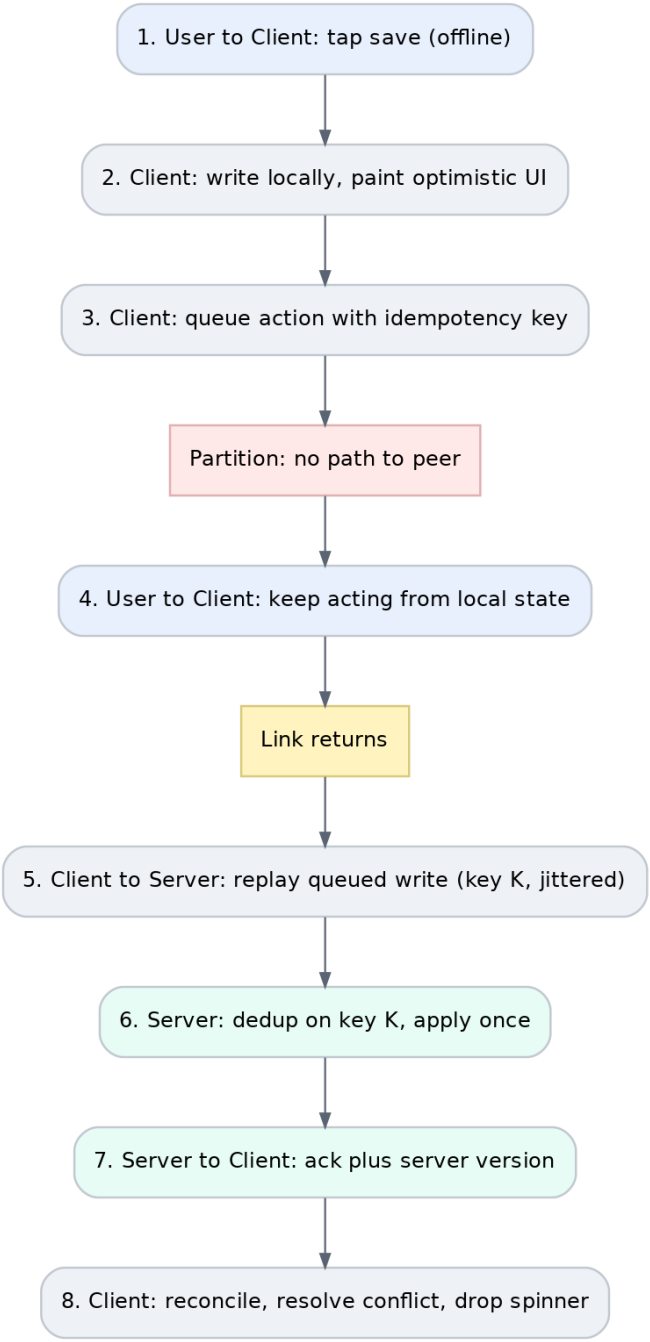


Figure 2: Tracing one write across a partition and back, from optimistic local save to server reconciliation

write but before the response reached the phone, the client genuinely cannot tell whether the write landed. Retry and you might charge the card twice. The foundation's fix is load-bearing and worth memorizing: make writes idempotent with a client-generated key. The client mints the key, the server deduplicates, and a retried write that already landed becomes a safe no-op. Idempotency is not a nicety on a flaky network; it is the only thing that makes retrying safe at all, and on the client retrying is unavoidable, so idempotency is mandatory. You will see this again in chapter 13 when we design offline-friendly APIs, and again in chapter 18 when we build resilience patterns into the client.

CAP, and why the client lives on the AP side

Now make it formal, because the formalism pays for itself the first time it tells you what your app must do before you have written a line.

The CAP theorem, in the precise form the foundation insists on, says this: during a network partition (P), you can have at most one of consistency (C) or availability (A). Brewer's own 2012 clarification matters and people forget it: the tradeoff only bites during a partition; the rest of the time you can have both. CAP is a starting point, not the whole story. The more useful daily framing is PACELC, Abadi's extension: if there is a partition, choose between availability and consistency; else, in the normal case, choose between latency and consistency. PACELC is the better tool because partitions are rare but the latency-consistency tradeoff is constant, and your mobile users feel latency every single day, every time a screen chooses to wait for the authoritative answer instead of showing the cached one.

Here is why this is the chapter's hinge. When the partition comes, a system has to choose: refuse to operate so it never serves a wrong answer (consistency), or keep operating and risk serving a stale one (availability). The client almost never gets to choose consistency, because choosing consistency during a partition means going dark, and an app that goes dark the moment the network drops is a broken app. So the client lives on the AP side. It stays available. It keeps showing data and accepting input from local state. And availability during a partition has a price the foundation names exactly: the client is, in effect, a replica that can go offline and fall arbitrarily far behind, so when the link returns, it owes a reconciliation. It must reconcile later. Every offline edit you accepted while partitioned is a write that has to be replayed, deduplicated, and merged against whatever the server did in the meantime.

This is the single most clarifying lens in mobile architecture, and the foundation hands you the habit that goes with it: name the consistency model before you design the feature. Before building any synced feature, state out loud what consistency the backend offers and what the user actually needs. The gap between those two is exactly the work, the optimistic updates, the conflict resolution, the staleness indicators. Most sync bugs come from never having named the model. You can say it in one sentence per entity, and if you cannot say the sentence, you have not designed the system, you have hoped.

The good news, and it is genuinely freeing, is that you almost never need strong consistency. You need it to feel consistent. The foundation is emphatic that Spanner-grade external consistency is expensive and almost never what a mobile feature requires; read-your-writes plus honest staleness, delivered through optimistic UI and good reconciliation, gives users the experience of consistency at a fraction of the cost. Read-your-writes is the session guarantee that you always see your own writes, and the mobile trick is that you usually fake it on the client with an optimistic update

rather than paying for it on the server. The user taps "like," you paint the heart red instantly from local state, and you reconcile with the server in the background. They experienced consistency. The system underneath was eventually consistent the whole time. Knowing the difference between feeling consistent and being consistent is an architect-level judgment, and it is the judgment that lets you ship fast apps on slow networks.

Reframe your local cache one more way and the right behaviors fall out for free. Your client cache is a follower replica of the server's data. It can be partitioned (offline), it lags (eventual consistency), and reading from it is reading from a stale replica. Once you name it that way, the foundation says, the correct behaviors are forced: show staleness honestly, prefer read-your-writes via optimistic updates, refresh on foreground, and never let the user believe the cached value is authoritative when it is not. The worst failure here is the quiet one, the stale read presented as fresh, where the client shows old data as if it were current and the user acts on it and is confused or harmed. Tag data with freshness. Design the UI to be honest about it. A node that lies about how current its data is, is worse than a node that admits it is offline.

It helps to model the client as a small state machine rather than a binary online or offline flag, because the interesting behavior lives in the middle. The node moves between three states, and each one has a different contract with the user:

Degraded is the state most apps forget. It is the place where requests still sometimes succeed but the latency is climbing and the error rate is up, which is exactly when a naive client retries hardest and makes things worse. A node that recognizes degraded as its own state opens its circuit breaker, leans on cache, and stops hammering, instead of treating every slow call as a fresh chance to pile on. The transition out of offline matters just as much: the link returning is not a green light to fire everything at once, it is the start of a careful drain, which is the storm problem we get to shortly.

The most hostile node in the system

Untrusted, on a flaky network, on a device you do not control, running a version you shipped months ago and cannot hotfix. That is the client, and each clause is a design constraint, not a complaint. Walk them.

Untrusted. The client runs on hardware the user owns and an attacker can own too. It can be inspected, modified, instrumented, and lied to. Anything you trust the client to enforce, an attacker can disable. This is why auth and security sit so high in the staff reasoning order, and we will spend real time on it in chapters 21 and 22, but the node framing is what makes it intuitive: a node you do not physically control is a node whose every claim you must verify on a node you do. The client can ask; the server must decide.

On a flaky network. We have made this formal already, but as a hostility property it means the client is the node most likely to be partitioned, by a wide margin. A backend node is partitioned when something is badly wrong. The client is partitioned every commute. Designing for the partition is not defensive over-engineering on the phone; it is the base case.

On a device you do not control. You do not pick the hardware, the OS version, the available memory, the battery state, or the dozen other apps fighting for the radio. The foundation's whole performance discipline, startup budgets, memory budgets, the binary that is a monolith no matter how modular your source is, exists because the device is a shared, hostile, varied environment and you ship one binary into all of it. The node runs everywhere, on the newest flagship and the four-year-old budget phone with a swollen battery, and it has to behave on both.

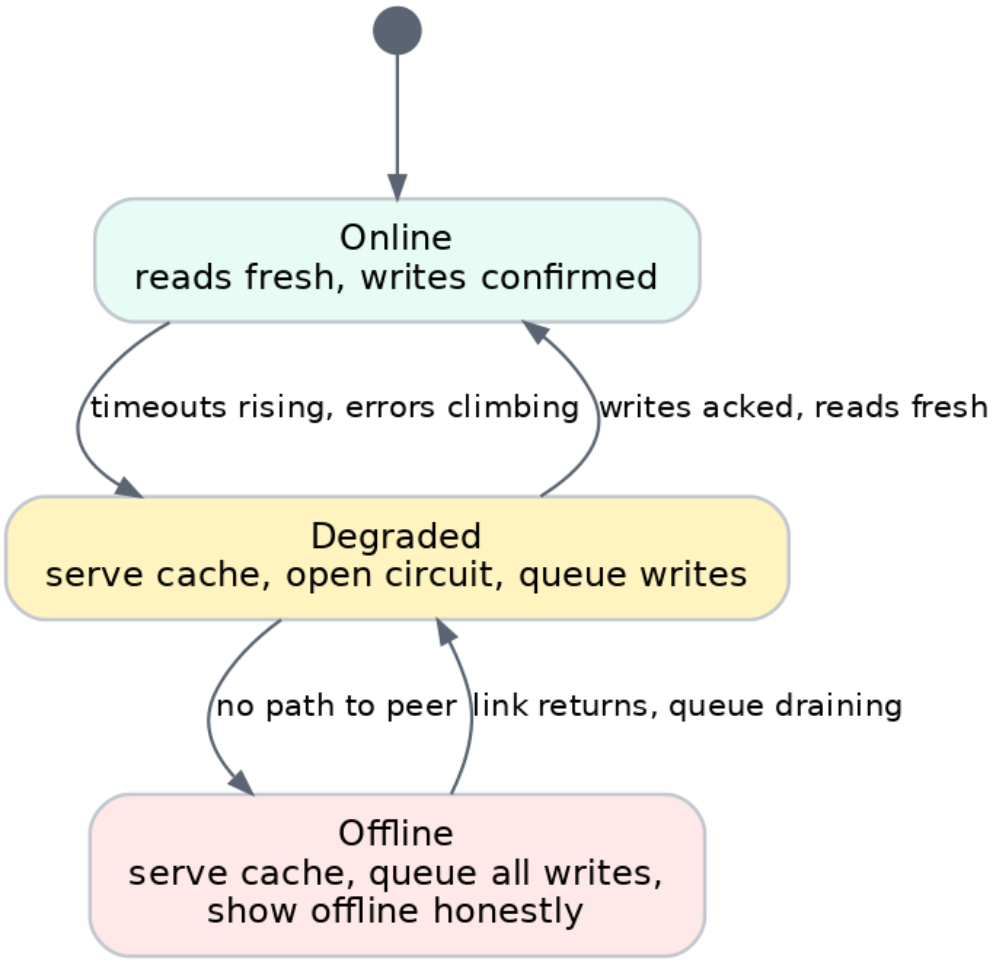


Figure 3: The client as a three state machine moving between online, degraded, and offline

The platform also enforces this hostility with a literal deadline. The OS does not politely wait while your node does too much work at the wrong moment. On iOS, a system watchdog terminates an app that takes too long to launch, with the launch watchdog limited to roughly twenty seconds (Apple, "Addressing watchdog terminations," developer.apple.com). That number is a hard upper bound, not a target, and the way you blow past it is almost always by treating launch as a place to do node work synchronously: a blocking network call to fetch config, a cache migration that walks every row, an auth handshake that has to round-trip before the first screen draws. The partition you keep forgetting reaches all the way into your launch path. If your app needs the network to start, then the elevator, the plane, and the dead zone are not just degraded experiences, they are launches that hang against the watchdog and get killed by the OS in front of the user. A node that cannot boot without its peer is a node that the platform will shoot for you.

And here is the clause that defines mobile and separates it from every backend system: running a version you shipped months ago and cannot hotfix. You cannot hotfix a phone. When you find a bug in a backend service, you fix it and redeploy, and minutes later every request is hitting the fixed code. When you find a bug in a shipped app, the fix is a new binary that goes through store review and then waits, for weeks, for months, on users to update, and some of them never will. Your installed base at any moment is a long tail of every version you have ever shipped, all live, all in production, none of them patchable. The foundation's framing of observability follows directly: it is telemetry you cannot hotfix, which means the instrumentation you forgot to ship is instrumentation you simply do not have until the next release reaches users. You design the binary to tell you what is wrong before you ship it, because after you ship it, the binary is all you get.

Put the four clauses together and the conclusion is unavoidable. The client is not a difficult node; it is the worst node, the one with the least trust, the worst connectivity, the least control, and the slowest path to a fix. And you run millions of copies of it. Which brings us back to the point we deferred.

The client is the load, and that makes you dangerous

The single most damaging thing a mobile client can do is turn a small backend hiccup into a large outage by retrying aggressively in lockstep with millions of peers. The foundation calls this out as the failure the client population uniquely causes and uniquely must prevent, and it is the reason a staff mobile engineer treats client retry behavior with the same rigor a backend engineer treats a rate limiter.

The mechanism is worth seeing in full, because once you see it you cannot unsee it. The backend has a brief outage, thirty seconds, say. During those thirty seconds, every instance of your app that tried to make a request failed. The instant the backend recovers, all of those clients retry. If they retry on a fixed timer, they retry together, a synchronized stampede, and the recovered backend, which has just barely come back, is instantly knocked flat by a wall of traffic larger than its normal peak. You have built a self-inflicted distributed denial of service, and your own app is the botnet. Worse, the caches that expired during the outage all expired together too, so they all stampede the database together, a thundering herd on top of the retry storm.

It can get worse than a single knockdown. The foundation describes metastable failure, where the system gets stuck in a bad state that sustains itself even after the original trigger is gone, because the load created by the failure, the retries, the cache misses, keeps it overloaded. The trigger was a thirty-second blip; the outage that follows can last hours, because the recovery traffic itself is now

the problem, and the only exit is shedding load, not waiting it out. If you have ever watched a brief incident refuse to end, this is often why. The client population turned a moment into an ordeal.

The defense is not exotic, and the foundation states it as the single most important resilience habit a mobile engineer can build: never retry without backoff and jitter. Exponential backoff grows the wait after each failure so you stop hammering. Jitter, the part that actually matters, randomizes the wait so your fleet does not retry in lockstep, because correlated retries where everyone backs off to the same instant just recreate the overload. Full jitter, a random wait up to the cap, is a solid default. Cap your retries, honor Retry-After headers, and stop retrying on errors that will never succeed, because a 400 is not going to become a 200 no matter how many times you ask. Add a client-side circuit breaker that stops calling a dead endpoint so it can recover, and stagger reconnection for any persistent connections. And apply the same jitter to cache expiry, so your caches do not all expire at the same instant and stampede together.

None of those techniques are hard. The hard part is the mindset that makes you reach for them, which is the node mindset, the recognition that you are not one client being polite, you are one of millions of identical nodes whose collective behavior is a force of nature aimed at your own backend. This is the most consequential distributed-systems lesson for a mobile fleet, and it is the cleanest illustration of why the reframe in this chapter matters. A screen does not cause outages. A node, multiplied by millions and behaving in lockstep, absolutely does. The discipline of jitter, bounded retries, and circuit breakers on the client is not politeness. It is the only thing standing between a small incident and a large one, and owning it is part of what it means to operate at staff level.

In practice: a feature that survives the bad day

Walk a real one end to end, because the node framing earns its keep only when it changes a design.

You are adding the ability to "save" an item, a bookmark, a favorite, the shape does not matter. The feature owner's version is two lines: call the save endpoint, show a filled star. It demos perfectly. Now design it as a node.

Start by naming the consistency model, the foundation's highest-leverage habit. A saved item is single-writer, low-stakes data: only this user saves their own things, and a conflict between two of their devices is rare and cheap to resolve. So the model is read-your-writes for this user, eventual consistency under the hood, and last-write-wins is acceptable for the rare device conflict as long as it is not arbitrated by a wall clock. That one sentence, said before any code, is the design.

Now the partition. The user taps save on the subway. The node is partitioned, so you do not wait for the server and you do not fail. You write to the local store first, paint the star filled immediately (the optimistic update that delivers read-your-writes), and queue the save as a pending action with a client-generated idempotency key. The user experienced an instant, consistent app. The truth is queued.

Now the reconnection, designed for the storm rather than the demo. When the link returns, you do not fire the queued save the instant connectivity flips, because so did a million other phones coming out of the same dead zone. You replay it under jittered backoff, with a bounded retry count, honoring any Retry-After the server sends. Because the write carries an idempotency key, a retry that the server already processed is a safe no-op, so the partial-failure case, where the save landed but the response was lost, cannot double-save. If the server rejects the write permanently with a 4xx, you stop retrying and surface the failure honestly rather than spinning forever.

Now the staleness honesty. When the user reopens the app the next morning, the star they see comes from the local follower replica, which may be stale if they saved something on another device overnight. So you refresh on foreground and reconcile, and if there was a conflict you resolve it by the rule you named up front rather than discovering you had no rule during a support escalation.

Two lines became a design with a named model, an optimistic write, an idempotent queued action, jittered reconnection, and foreground reconciliation. Every one of those fell out of treating the save as a node operation rather than a button. That is the payoff of the reframe, and it is the difference between a feature that works in the demo and one that works on the train.

What a staff engineer decides

- Name the consistency model before writing code, one sentence per synced entity: where the source of truth lives, the sync direction, and the conflict policy. If you cannot say the sentence, you have not designed the feature.
- The client lives on the AP side of CAP. It stays available during a partition and therefore owes a reconciliation when the link returns. Design the reconciliation up front, not in the postmortem.
- You almost never need strong consistency; you need it to feel consistent. Deliver read-your-writes with optimistic updates and honest staleness rather than paying for server-side linearizability.
- Treat the local cache as a stale follower replica. Show staleness honestly, refresh on foreground, and never present eventually-consistent data as if it were authoritative.
- Every client write that can be retried must carry a client-generated idempotency key, because on a flaky network retrying is unavoidable and double-applying is unacceptable.
- The client is the load, so the client owns the blast radius. Jittered backoff, bounded retries, and a client circuit breaker are not politeness; they are the line between a small incident and a large one.
- Design for the partition, not the demo. The happy path on office wifi is the one environment your users are rarely in.

Common interview questions

- State the CAP theorem precisely and explain why PACELC is more useful in practice. A strong answer covers that during a partition you get at most one of consistency or availability, Brewer's clarification that the tradeoff only bites during a partition, and PACELC's else case, the everyday latency-versus-consistency choice users actually feel.
- Where does the mobile client sit on the CAP spectrum, and what does that obligate it to do? A strong answer puts the client on the AP side because going dark during a partition is a broken app, and names the obligation that follows: reconcile queued offline writes against the server on reconnect.
- You have a million clients and the backend has a thirty-second outage. What happens on recovery if clients retry naively, and how do you prevent it? A strong answer describes the synchronized retry storm plus simultaneously expired caches as a metastable, self-sustaining overload, and prescribes full jitter on retries and cache expiry, bounded retries, a client circuit breaker, and staggered reconnect.
- Why must client writes be idempotent, and whose job is the key? A strong answer explains that a dropped response makes the client unable to tell whether a write landed, so retries can

double-apply, and that the client mints an idempotency key the server deduplicates against.

- What is the difference between latency, partition, and partial failure from the client's seat? A strong answer treats them as three states with three responses: timeout-and-degrade for latency, work-from-local-state for partition, and compensating actions (a saga) for partial failure.
- Why can you not treat the mobile client like any other node in the system? A strong answer names the four hostility properties, untrusted, flaky network, uncontrolled device, unpatchable shipped version, and draws the consequence that you instrument and harden the binary before ship because you cannot hotfix a phone.
- Why is a chatty client expensive in a way a chatty backend service is not? A strong answer explains the cellular radio's RRC state machine and the tail: each request wakes the radio and leaves it in a high-power state for several seconds afterward (network-configured, up to about fifteen), so frequent small calls drain the battery far beyond the time the transfers take, which makes batching and coalescing first-order design on mobile, not a micro-optimization.
- What happens if your app needs a network round trip to finish launching? A strong answer connects the partition to the platform's launch watchdog: iOS terminates a launch that runs too long (roughly twenty seconds), so a client that blocks startup on the network hangs and gets killed by the OS in dead zones, which is why launch-critical work must come from local state with the network deferred.

Key takeaways

- The client is a node, not a screen: it holds state, gets partitioned, and must behave correctly when it cannot reach its peers and they cannot reach it.
- It is the most hostile node in the system, untrusted, on the worst network, on a device you do not control, running a version you cannot hotfix, and you run millions of copies at once.
- The client lives on the AP side of CAP, so it stays available during a partition and owes a reconciliation afterward. Name the consistency model before you design the feature.
- You rarely need real consistency; you need the experience of it, delivered with optimistic updates, honest staleness, and good reconciliation.
- Idempotency keys make retrying safe, and on a flaky network retrying is unavoidable, so they are mandatory.
- The client is the load. Jittered backoff, bounded retries, and circuit breakers are the only thing keeping your own fleet from turning a hiccup into an outage. Design for the partition, not the demo.

Exercises

Do these with a pen, not a keyboard. The point is to make the client-as-node lens reflexive, so that you reach for it before you reach for an endpoint.

Warm-up. Take a feature you shipped recently that does a single network write, a like, a follow, a "mark as read," anything. Write the one-sentence consistency model for it: where the source of truth lives, the sync direction, and the conflict policy. Then list, in order, what your current code does when the user triggers it on the subway with no signal. If the honest answer is "shows a spinner and eventually an error," you have found a screen pretending to be a node.

Build. Design the offline behavior for a "move money between my own accounts" transfer in a

banking app, end to end, as a node. Name the consistency model. Decide what the client does when the user taps confirm with no connectivity, what it queues, what it shows, and what carries the request across the partition so a retry cannot move the money twice. Decide what the client must refuse to do offline and why. Sketch the state machine of the transfer from tap to confirmed, including the ambiguous state where the request was sent but never acknowledged.

Stretch. Your app has eight million installs. A backend region fails over at 8:00am local time, right at the morning commute peak, and is unreachable for forty seconds. Write the incident you would expect to see if every client retried on a fixed three-second timer, then redesign the client so the same failover is a non-event. Be specific about the numbers you can reason from: the radio tail that each retry leaves behind, the jitter window you would choose and why, what the circuit breaker watches, and how foreground reconciliation behaves when eight million apps come back at once. Then name the one telemetry signal you would have wanted in the binary before this happened, given that you cannot hotfix a phone.

Worked solutions

Warm-up. Take a “mark as read” on a message. The one-sentence model: the source of truth is the server, the sync direction is client to server with the read state owned by this single user, and the conflict policy is last-write-wins on a server sequence because read state is single-writer and low-stakes. On the subway with no signal, the node version is short. Write read=true to the local store, update the UI immediately from local state, and queue the change with an idempotency key. There is nothing to spin on, because marking something read is not a request that has to succeed before the user can move on, it is a local fact you will sync later. If your shipped code instead awaited the network and showed a spinner, the fix is not a better spinner, it is moving the write to local-first and letting reconciliation happen in the background. The lens did its job the moment it turned a network call into a local write plus a queued sync.

Build. The transfer is the interesting case because the instinct, work offline, is partly wrong, and naming the model is what tells you where. The model: the source of truth is the server, full stop, because this is money and the server is the only node allowed to decide the balance moved. The sync direction is client to server only, and there is no client-side conflict resolution because the client is never authoritative about a balance. That single sentence already decides the hard part. The client does not get to “complete” a transfer offline, because a node that cannot see the authoritative balance cannot honestly tell the user the money moved. So when the user taps confirm with no connectivity, you do not optimistically show “transferred.” You capture the intent, mint a client-generated idempotency key, queue the request, and show an honest pending state: “this transfer will be submitted when you are back online.” That is the line a feature owner gets wrong, because the optimistic-UI habit that is right for a like is a lie for a balance.

Carrying the request across the partition is the idempotency key plus the server’s deduplication, and here the ledger gives you a concrete window to design against: Stripe, for instance, deduplicates on the Idempotency-Key header but only retains keys for twenty-four hours (docs.stripe.com), so a transfer that sits queued on a phone for two days offline cannot rely on the server still remembering the key. The node behavior that follows is to treat a very old queued write as needing re-confirmation, not blind replay, because the safety net has expired. The ambiguous state, request sent but never acknowledged, resolves the same structural way every money transfer resolves it: the client does not guess, it asks the server for the status of that idempotency key, and the server, which applied the write at most once, returns the truth. What the client must refuse to do offline

is anything that depends on a number only the server knows: it will not show a new balance, it will not allow a second dependent transfer against assumed funds, and it will not claim success. The state machine runs tap, then local-pending-with-key, then on reconnect submitted, then either confirmed-by-server or rejected-by-server, with the sent-but-unacknowledged branch routing to a status query rather than a retry. The whole design fell out of one honest sentence about where the truth lives.

Stretch. The naive incident is the chapter's worst case made arithmetic. Forty seconds of outage means roughly thirteen retries per client on a fixed three-second timer, all of the fleet's clocks aligned to the failover instant, so the moment the region comes back it takes a wall of traffic larger than its normal morning peak and goes straight back down, and the caches that expired during those forty seconds all stampede the database in the same instant. That is a metastable state: the recovery traffic is now the load, and waiting does not help because the load is self-sustaining. There is a second-order cost the radio tail makes concrete. Every one of those synchronized retries wakes the cellular radio and leaves a tail of several seconds, up to about fifteen, network-dependent (ACM MobiCom 2015 RRC study), so the naive design is not only re-breaking your backend, it is draining eight million batteries to do it.

The redesign makes the same failover boring. Retries use full jitter, a random wait up to a growing cap, so the fleet does not converge on any instant; with a cap in the tens of seconds, eight million reconnects spread across a wide enough window that the recovered region sees a ramp rather than a spike. Retries are bounded, so a client that keeps failing stops rather than hammering, and the client opens a circuit breaker when the error rate and latency cross a threshold, watching the degraded state from the state machine so it leans on cache instead of piling on. Cache expiry gets the same jitter so the caches do not all miss together. Foreground reconciliation, when the commuters reopen their apps, refreshes against a server-issued cursor rather than a wall clock, and it too is staggered by jitter so the read stampede is spread. The radio cost drops out of this for free: fewer, jittered, batched retries mean fewer wake-ups and fewer tails. The one telemetry signal you wish you had shipped is a client-side measurement of retry attempts and circuit-breaker state per session, reported once connectivity returns, because that is the number that tells you whether your fleet amplified the incident, and you cannot add it after the fact. You design the binary to tell you it behaved, before you ship the binary, because the binary is all you get.

Further reading

The sources behind this chapter, all from the research foundation:

- Designing Data-Intensive Applications. Martin Kleppmann, O'Reilly, 2017. https://dataintensive.net/
- "Timeouts, retries, and backoff with jitter." Marc Brooker, Amazon Builders' Library. https://aws.amazon.com/builders-library/timeouts-retries-and-backoff-with-jitter/
- "Dynamo: Amazon's highly available key-value store." DeCandia et al., SOSP 2007. https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf
- "CAP and PACELC: thinking more clearly about consistency." Marc Brooker, 2014. https://brooker.co.za/blog/2014/07/16/pacelc.html
- "CAP twelve years later: how the rules have changed." Eric Brewer, 2012. https://www.infoq.com/articles/ca twelve-years-later-how-the-rules-have-changed/
- "Caches, modes, and unstable systems." Marc Brooker, 2021. https://brooker.co.za/blog/2021/08/27/caches.
- "Retry storm antipattern." Microsoft Azure Architecture Center. https://learn.microsoft.com/en-us/azure/architecture/antipatterns/retry-storm/

- "Local-first software: you own your data, in spite of the cloud." Ink and Switch, 2019. https://www.inkandswitch.com/essay/local-first/
- "How we made Notion available offline." Notion, 2025. https://www.notion.com/blog/how-we-made-notion-available-offline
- RRC state machine and tail energy (ACM MobiCom 2015 study, Stoner et al.). https://www.sigmobile.org/mobicom/2015/papers/p477-stonerA.pdf
- "Addressing watchdog terminations." Apple, developer documentation. https://developer.apple.com/docume watchdog-terminations
- Idempotent requests. Stripe API documentation. https://docs.stripe.com/api/idempotent_requests

You can now name the model and design for the bad day. But naming a model is one decision among dozens in any real design, and a staff engineer is judged on whether they can evaluate a whole architecture, surface its trade-offs, and write the decision down so the next person knows why. That is the skill of reading an architecture, and it is chapter 3.

Chapter 26: Case study, a ride-hailing platform

A driver accepts your ride. For the next forty minutes, a car moving through real traffic, a phone in your pocket, a phone on the driver's dashboard, and a fleet of backend services in three data centers all have to agree on one small fact: where the car is and what the trip is doing. They have to keep agreeing through a dropped tunnel signal, a region failover, and the moment a stadium empties and a hundred thousand people open the app at once. Nobody gets to call a meeting. The agreement has to fall out of the architecture.

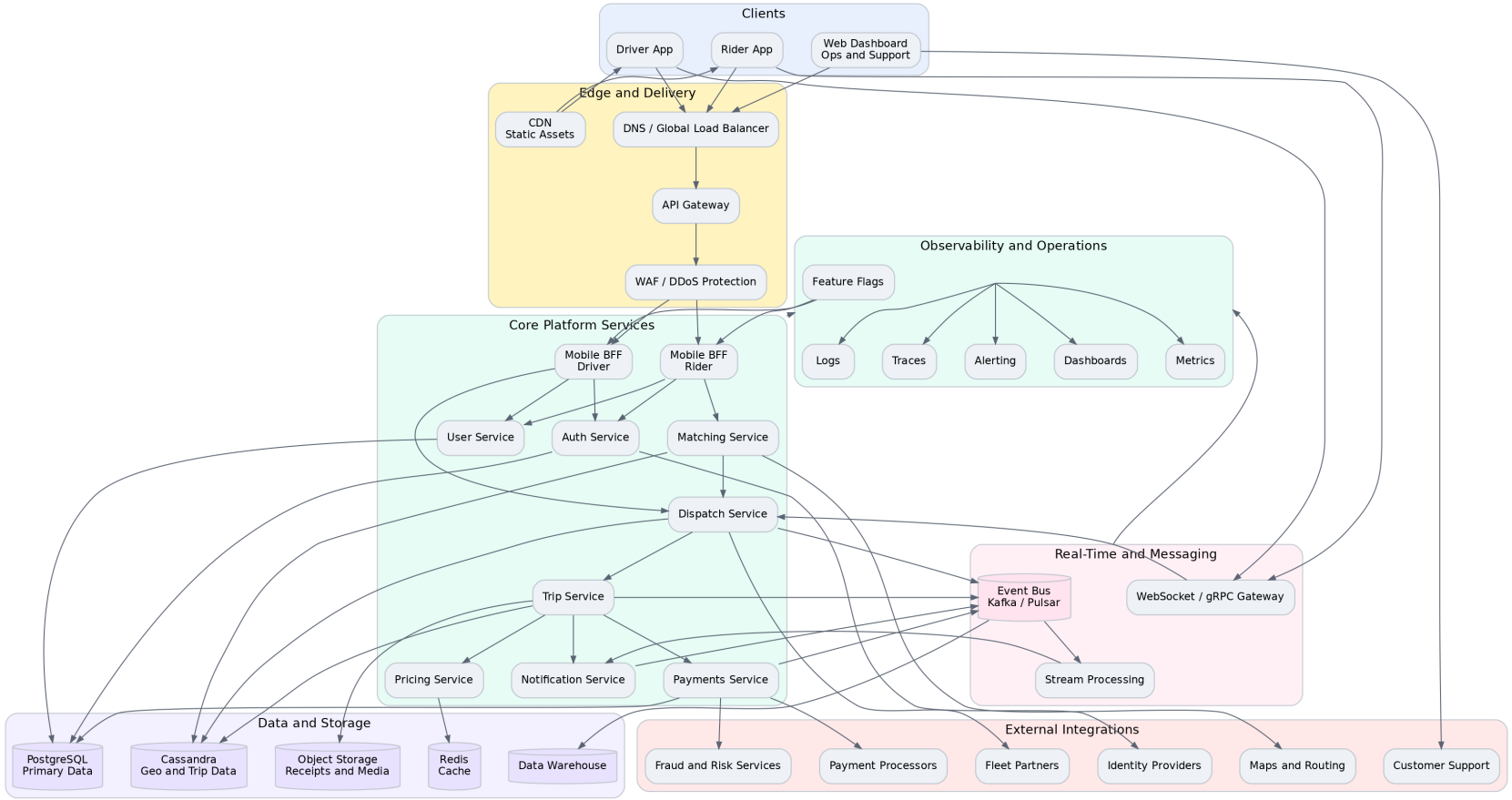
That is the problem this chapter takes apart. A ride-hailing app looks like a map with a moving dot, and underneath it is one of the hardest consumer distributed systems ever shipped to a phone: a real-time, multi-party state machine running across nodes that lie about their clocks, drop off the network without warning, and reconnect in stampedes. If you can reason about this system, you can reason about almost any client you will ever own.

A note before we start. This is a case study, not an exposé. What follows is a representative architecture for a ride-hailing client, assembled from public engineering talks, papers, open-source projects, and platform documentation from companies including Uber and others. It is a teaching model, not a claim about any company's current internal systems. Where a figure or a design is publicly documented, this chapter cites it with a real source. Everything else is illustrative: a plausible design chosen to make a principle concrete, not a leaked schematic. When you read "a representative ride-hailing client structures its app this way," read it as "here is a defensible way to build this, and here is why," not "here is what is running in production at company X tonight." The reasoning transfers. The unverifiable internal details do not.

By the end of this chapter you can:

- Decompose a real-time, multi-party mobile product into an architecture you could defend in a staff design review, including the client app structure, the mobile BFF, the dispatch state machine, and the push platform.
- Design a client-side trip state machine and an offline reconciliation protocol that survives disconnects without losing or duplicating updates, using server-issued monotonic versions rather than client clocks.
- Read a published push-platform evolution (polling, server-sent events, then gRPC) and explain the trade each generation made, separating the documented facts from the parts you should treat as illustrative.
- Write the architecture decision record and the incident post-mortem that a staff engineer owns when this system is theirs.

Before the detail, here is the entire ride-hailing platform on one page. Treat it as the map for this chapter. Every section that follows zooms into one part of it, and the prose refers back to these components by name rather than redrawing them.



A representative ride-hailing platform, the complete end-to-end architecture

The shape of the problem

Start concrete. A rider opens the app, requests a ride, watches a car approach, takes the trip, and pays. A driver receives an offer, accepts, drives to the pickup, runs the trip, and drops off. Those two stories share one trip object, and that object lives in at least four places at once: the rider's phone, the driver's phone, the matching backend, and the trip-storage backend. Every copy can be stale, every link can fail, and the user is staring at a map that has to feel live the entire time.

This is the recurring spine of the whole book, so name it again. The client is a node, not a screen. A ride-hailing client is the clearest case you will find: the phone is a replica of a trip record, holding the stalest copy, on the flakiest link, with the least trustworthy clock, and yet the product demands it feel authoritative. Hold that frame and the design decisions stop looking like app-developer choices and start looking like distributed-systems choices, because that is what they are.

Three constraints drive everything. The experience must feel real-time, so polling on a timer will not do. The core flow, request a ride, take it, pay, must be close to always available, because a failure here is not a missed feature, it is a person stranded at the curb. And the client population is enormous and correlated, so when the network hiccups, millions of phones react in the same instant, and the client becomes the load that either drains an incident or amplifies it. That third constraint comes back to bite in the post-mortem.

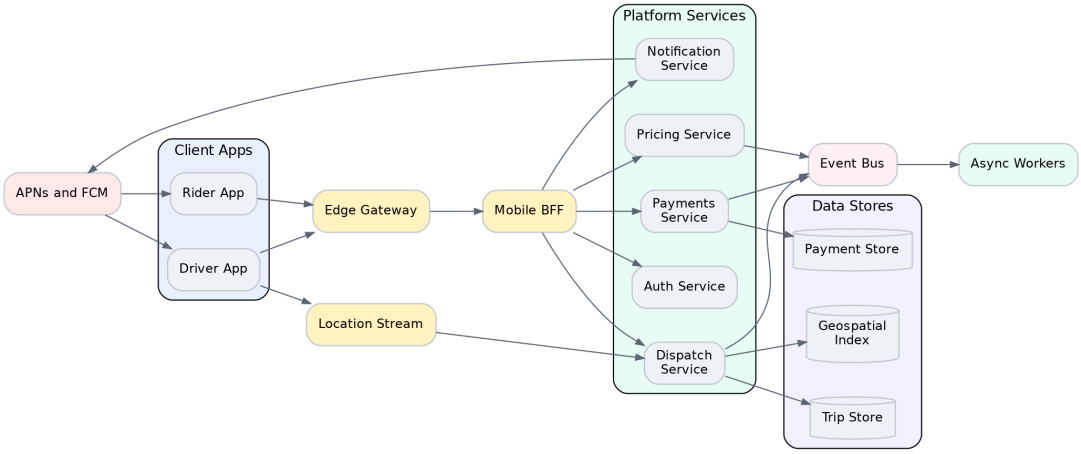


Figure 4: A representative ride-hailing platform, end to end

What the business asks for, and why each ask bends the architecture

Before any diagram, get the requirements straight, because in this product the requirements are not a checklist of features. Each one bends the architecture in a specific direction, and a staff engineer can name which bend belongs to which requirement. Walk the product surface the way a rider experiences it, and watch the system fall out of the experience.

A rider requests a ride. That single tap has to be near instant and near always available, because the request is the front door to the entire business. So the request path cannot depend on anything optional. It runs through the core flow, gated behind the stricter review the core-optional split

exists to enforce, and it tolerates partial degradation of everything around it: the promo banner can fail to load, the request button cannot. The requirement that requesting a ride must work even when the surrounding app is wobbling is what forces the core-optional boundary to be mechanical, not a convention. You will see that boundary again in the client structure.

The platform matches the rider to a driver. Matching is a spatial, real-time optimization over a moving fleet, so it cannot live on the phone and it cannot be a simple database lookup. It needs the driver fleet's live positions, indexed spatially, and a matching service that runs continuously, not on request. That requirement is why a dispatch service exists as its own tier with its own data model, and why the client is a thin participant in matching rather than a driver of it. The phone proposes "I want a ride here"; the backend decides who gets it.

Dispatch hands the trip to a specific driver and holds the offer open for a few seconds. A driver offer is the most time-sensitive message in the system: a 30-second offer that arrives 8 seconds late is a bug that costs a match. So the delivery channel for offers cannot be best-effort. It needs acknowledgment, it needs to measure round-trip time, and it needs to resend reliably inside a tight deadline. That single requirement, deliver a time-boxed offer reliably, is most of the reason the push platform evolved the way it did, away from a channel that could not tell a slow acknowledgment from a real loss.

Both phones track the trip live. Live tracking is the requirement that kills polling. A map with a moving dot that updates every five seconds feels broken, and a map that polls every second to feel smooth drains the battery and floods the gateway. So the architecture inverts: the server pushes location and state changes the instant they happen, over a connection held open for the duration of the trip, and the client renders interpolated motion between updates rather than demanding a fresh fix each frame. Live tracking is why there is a persistent real-time layer at all.

The rider pays, automatically, at the end of the trip. Payment is the one place in this system where you want consistency, not availability: a double charge or a lost charge is worse than a slow charge. So payment does not run on the optimistic, eventually-consistent path the rest of the trip uses. It runs through an idempotent, server-authoritative flow where the client's job is to display the result, not to decide it. The requirement that money must be exactly right, once, is what carves payment out as a different consistency regime from trip state. Hold that contrast: most of the trip is AP and optimistic, payment is the CP island inside it.

The system notifies both parties at the moments that matter: driver assigned, driver arriving, trip started, receipt ready. Notifications have to reach the user whether the app is foreground, backgrounded, or killed, which means they cannot ride only on the in-app connection. They need the operating system's push channels, APNs on iOS and FCM on Android, as a second delivery path that works when the app is not running. The requirement that a backgrounded rider still learns their driver arrived is why there are two real-time channels, an in-app stream and OS push, not one.

And the system prices dynamically. Surge pricing raises the fare when demand outruns supply in an area, which sounds like a pricing detail and is actually a system-shaping force, because surge is exactly when load spikes. A stadium empties, demand surges, the price surges, and the same surge that the pricing service computes is the surge of traffic that hits dispatch, the push platform, and the gateway all at once. The requirement to price for surge is inseparable from the requirement to survive surge. The moments your pricing model cares about are the moments your reliability model is tested, which is why correlated load is a first-class concern in every tier below.

Notice what just happened. Seven product requirements produced, between them, the core-optional split, a dedicated dispatch tier, a reliable time-boxed delivery channel, a persistent push layer, a separate consistency regime for money, a second OS-level notification path, and a standing assumption that load is correlated and spiky. The architecture is not imposed on the product. It is the product's requirements, taken seriously, one at a time.

The system on one path, request to receipt

Now trace one request end to end, naming each component by its responsibility, so the rest of the chapter has a shared map. As shown in the end-to-end architecture diagram, the components sit in a line from the phone to analytics, and a single ride request touches most of them.

The user taps request. The mobile app does not call the backend mesh directly. It assembles one request, the rider's location, the chosen product, the payment method, and hands it to the BFF, the mobile backend for frontend. The app's responsibility ends at "express intent and render truth": it holds the local trip state, drives the UI from it, and never tries to be the authority on what the trip is.

The BFF receives the request and is the client's single conversational partner. Its responsibility is aggregation and shaping: it fans out to the services the request needs, collects their answers, prunes them to what the screen binds to, and returns one mobile-shaped response. For a ride request, it forwards the intent to dispatch and returns immediately with a trip in the requested state, so the UI can move before matching completes.

Behind the BFF sits the API gateway and the service mesh. The gateway's responsibility is the edge: authentication, rate limiting, routing, and terminating the client connection. It is also, historically, where polling load showed up first, which is why the published account measured polling as a share of gateway traffic. The gateway is the choke point that makes wasteful client behavior visible.

Dispatch is the heart of the trip. Its responsibility is matching and the authoritative trip state machine: it holds the live fleet positions, runs the matching loop, selects a driver, issues the offer, and advances the trip through its states as the physical world advances. Dispatch is the single source of truth for "what state is this trip in," and every other copy, both phones, the BFF cache, is a replica of dispatch's answer.

The driver services are dispatch's counterpart on the supply side. Their responsibility is the driver's session, availability, and the firehose of location updates streaming up from every online driver's phone. Dispatch reads fleet position from here and writes offers to here.

Pricing computes the fare, including surge. Its responsibility is to turn a route and a demand picture into a number, and to expose surge multipliers that dispatch and the BFF read. Pricing is mostly a read dependency for the request path and a heavy compute dependency in the background, watching supply and demand per area.

The event bus carries state changes between services asynchronously. Its responsibility is decoupling: when dispatch advances a trip, it does not call notifications and analytics synchronously, it emits an event, and the interested services consume it. The bus is what lets the trip state machine run fast while slower consumers catch up at their own pace. This is the event-driven backbone chapter 19 builds, landing in a product.

Notifications consume trip events and deliver them to humans. Their responsibility is fan-out across channels: the in-app real-time stream when the app is connected, and OS push through APNs and

FCM when it is not. Notifications is where a single “driver arrived” event becomes a banner on the rider’s locked phone.

Analytics consumes the same events plus the client telemetry stream. Its responsibility is the record: every trip transition, every connect and reconnect, every reconciliation outcome, flowing into the metrics and tracing backend that tells you, after the fact, what the fleet actually did. You cannot attach a debugger to ten million phones, so analytics is the only eyes you have.

That is the whole path: user, app, BFF, gateway and mesh, dispatch, driver services, pricing, event bus, notifications, analytics. The rest of this chapter zooms into the parts of that path where the hard engineering lives, and it refers back to these components by the responsibilities just named.

The rest of this case study, and chapters 3 through 28, are in the full edition.

About the author

I'm Mike Salari.

For more than fifteen years, I've built software across Apple, Adobe, Cisco, Visa, Mastercard, and startups.

Altitude is a collection of the architectural patterns, systems-thinking principles, trade-offs, and lessons I've learned while building products at scale.

If this book helps you think more clearly, design better systems, and become a stronger engineer, then it has done its job.

salari.dev mike@salari.dev

Get the full edition

That was the sample. The complete Altitude is 28 chapters across seven parts, roughly 165,000 words, with 28 architecture diagrams, three full-page end-to-end case-study blueprints, worked exercises, a glossary, and a cited research foundation behind every claim.

If the first two chapters were useful, the rest goes further: the full contract with the backend, data and sync and resilience, real-time delivery, authentication, mobile security and threat modeling, observability, the cloud and Kubernetes mental model, and three complete case studies built from public engineering sources.

How to get it:

salari.dev

Message me on LinkedIn: linkedin.com/in/mike-salari

mike@salari.dev

Altitude, a field guide to staff-level mobile systems engineering. First edition, 2026. By Mike Salari.