SECOND EDITION

ALTITUDE

A FIELD GUIDE TO STAFF-LEVEL MOBILE SYSTEMS ENGINEERING



REAL-WORLD SYSTEM DESIGN

UBER • WHATSAPP • REVOLUT

+ 7 MORE TECH GIANTS

MIKE SALARI

FREE SAMPLE OF ALTITUDE

Staff mobile engineer is not senior plus

What the staff job changes, week by week, and what to start doing before anyone gives you the title.

MIKE SALARI

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

A free sample of *Altitude: A field guide to staff-level mobile systems engineering*, second edition, version 2.1. An essay on the staff role, one section from chapter 22, a Companion worksheet, and what the full book covers.

SALARI.DEV

THE ROLE

A promotion that changes the job

A senior engineer is responsible for their own work. A staff engineer is responsible for everyone's. It sounds like a small change in wording, and most people treat it that way when they get the title. Then the first quarter goes badly and they can't explain why.

The usual story goes like this. A company promotes its best senior mobile engineer, the one who fixes the strange crash at six in the evening, knows where every old workaround lives and merges clean code all week. Someone tells them to keep doing what they do, just more of it. So they take more tickets, answer more Slack threads, review more pull requests. Six months later they are the most tired person on the team, the backlog is no shorter, and the promotion feels more like a fine than a reward.

Nothing is wrong with the engineer. They're using the habits that made them a strong senior in a job that needs different ones. There are thirty developers now, on iOS and Android, and no amount of extra evenings makes one person's output cover thirty.

THE ROLE

The part nobody puts in the offer letter

A staff engineer has no one reporting to them. When the networking layer or the module structure needs to change, they can't assign it and walk away. They have to convince the engineering manager it matters this quarter, convince product it's worth the delay, and convince the developers doing the work that the approach makes sense. Every decision goes through people who can say no.

So the job has two halves that rarely appear together in a job post. One is technical judgment: knowing which change saves the team a year and which one only looks clever. The other is the conversation around it: knowing when to push, when to give ground, and how to get a yes from someone who owes you nothing.

What the work looks like across a week

Monday

On Monday you check two take-home assessments. One is tidy code that misses what the task was testing. The other is rough but handles the case where the phone loses its connection halfway through. You'd hire the second person, and now you have to write feedback that explains why to an interviewer who liked the first one.

> WHAT A SENIOR CAN TAKE FROM THIS
>
> Ask to review a take-home assessment or sit in on an interview. You'll learn what your company values much faster than from any career framework.

Tuesday

On Tuesday you meet the engineering manager. They own the people, you own the technical direction, and the meeting is about where those two overlap: who is stuck, what is likely to slip, what the team should stop doing. Afterwards you hand out work. One developer takes the networking layer, another takes the migration, and each of them hears why the piece is theirs.

> WHAT A SENIOR CAN TAKE FROM THIS
>
> Next time you split a task with someone, take two minutes to explain why each person gets their part. People notice.

Wednesday

On Wednesday you sit with the other staff engineers, backend, iOS and Android, and agree one API contract, so both apps read the same data the same way and the same business rule isn't written twice by two teams that rarely talk. Everyone in the room is capable and everyone has an

opinion. What carries the decision is a clear account of the trade-offs, because thirty people will work inside it for the next two years.

> WHAT A SENIOR CAN TAKE FROM THIS
>
> Before your next design discussion, write half a page with the options and what each one costs. The person who brings the page usually shapes the decision.

Thursday

On Thursday something breaks for the business. It might be a crash spike after an iOS release, or ANRs climbing on budget Android phones after a change to app startup. You're expected to be first on the line, the way an officer is at the front, and the reason has nothing to do with whose code it was. Someone has to understand the whole system and keep the room calm while the fix is found.

> WHAT A SENIOR CAN TAKE FROM THIS
>
> During the next incident, stay for the whole thing, including the write-up afterwards. Most people leave once their part is fixed. The write-up is where you learn how the system fails.

Friday

On Friday you may be at a client's office, speaking for the mobile team and sometimes for the company.

> WHAT A SENIOR CAN TAKE FROM THIS
>
> Offer to join one call with a client or another department. Listen to what they ask about. It's rarely what engineers expect.

Between all of that sit the conversations that don't fit on a calendar: telling product a launch date won't hold, telling a strong senior their design won't survive fifty modules, choosing between two reasonable approaches knowing half the room would have picked the other one. Very little of the week is spent writing Swift or Kotlin.

THE EVIDENCE

What the job posts ask for

Real staff mobile job descriptions keep repeating the same list. Set technical standards across several teams. Own the roadmap and the architecture decisions. Review designs and code. Mentor engineers. Run technical interviews and decide on candidates. Take production issues and find the root cause. Make the whole team faster.

On iOS that usually means module structure, the move to Swift Concurrency and keeping up with what Apple changes every June. On Android it means Gradle builds that don't eat half the morning, app health in Google Play Console, and background work that survives phone makers who kill it to save battery.

None of the posts asks for the person who ships the most features. That's the clearest sign the role measures something else. Fox, Uber and Square each ask for technical standards, architecture or technical direction, and mentoring, and Uber adds raising the hiring bar through interviewing. The five postings, with quotes and links, are in the notes below.

THE HABITS

Where the senior plus habit shows

The difference is easiest to see in situations every mobile team has.

When the iOS team argues about SwiftUI and UIKit, the senior plus habit is a long Slack message explaining which one is better. It collects reactions and changes nothing. The staff habit is a one-page set of rules for mixing the two, so people can ship on Monday and move screens over gradually.

When product asks for a new SDK, the senior plus habit is to add it, see it build and merge. The staff habit is to check what it pulls in, how much it adds to app size and launch time on both platforms, and to go back to product with those numbers and a cheaper option.

When ANRs on Android climb and nobody can explain them, the senior plus habit is to move a suspicious block to a background thread and hope

the crash dashboard goes quiet. The staff habit is to go through the whole startup path, take disk work off the main thread, and add checks so the rate stays well under the 0.47% threshold where Google Play can show the app to fewer people.

When requirements change for the third time in a sprint, the senior plus habit is frustration in standup. The staff habit is to explain the cost in terms the business already tracks: release dates, crash-free sessions, users who leave.

In each case the senior solves the problem in front of them. The staff engineer also removes the reason it keeps coming back.

THE PATH

Getting there from senior

People rarely receive the title first and grow into it afterwards. More often they start doing the work, and the title follows once enough people have seen it.

Start by looking past your own team. Find a problem that hurts several teams and that nobody owns, such as slow builds, flaky UI tests or a module everyone avoids. Write a short plan, get agreement, and fix it with others rather than alone.

Keep a record of what changed because of you, with numbers where you have them. Promotion discussions happen months after the work, and nobody remembers March unless it's written down.

Build relationships with managers, product, backend and the other mobile platform before you need their support. That kind of trust takes a year or two to build, and it can't be borrowed on the day of a hard argument.

Stop being the person who unblocks everyone by doing the work. It looks generous, but it turns you into a bottleneck. Teach people until they no longer need you for daily problems.

Ask to write assessments and join interviews. Deciding who joins the team shows you, faster than anything else, what the company values.

THE TEST

A simple test

Imagine a month away with your laptop switched off. If the team keeps shipping, keeps the decisions you helped make and works from the documents you wrote, you're already doing staff work. If everything waits for you to come back, the title has changed and the job hasn't.

NOTES AND SOURCES

Five staff mobile postings, read on 27 Oct 2026. Quotes are the words on the page. Four of the five had been closed or removed by then; the text was still readable. Not every posting asks for every item.

Fox Corporation, Staff iOS Engineer. *"Establish engineering standards for architecture, automated testing, CI/CD, release management, and code quality"*; *"Drive the technical strategy for the mobile platform, including modularization, build system performance, developer tooling"*. foxcareers.com

Uber, Staff Software Engineer - iOS. *"raise the hiring bar through high-quality interviewing and feedback"*; *"Experience improving engineering standards across teams"*. jobs.anitab.org

Square (Cash App), Staff iOS Engineer, Neighborhoods. *"Guide Neighborhoods mobile technical direction across Cash App and Square"*; *"mentoring engineers and raising technical standards across the team and partner teams"*. builtinla.com

Capital One, Staff Android Engineer. *"Own technical design, architecture, and development of cross-functional, multi-platform application systems"*; *"Provide mentorship to Junior, Intermediate, and Senior mobile associates"*. builtin.com

The Washington Post, Staff Software Engineer, Android App Development. *"Conduct code reviews and ensure code quality"*; *"Contribute to the long-term technical roadmap for the Android platform"*. builtin.com

On the 0.47% figure. Google defines a user-perceived ANR rate of at least 0.47% of daily active users across all device models as bad behavior, and says an app above a bad behavior threshold is likely to be less discoverable on Google Play. Sources: Play Console Help, "Android vitals" and Android Developers, "Android vitals", accessed 27 Oct 2026. Altitude chapter 9 sets the full threshold table and says to budget below it with margin.

THE BOOK

What Altitude teaches

The essay was about the job. Altitude is about the work inside it: designing mobile systems at staff level, and defending the design when someone pushes back.

It starts from one fact. A phone is a node in a distributed system that you cannot patch, on a network you do not control. Chapter 2 builds on that, and the chapters after it ask the same questions of one layer at a time:

module structure, the API and the BFF, caching and sync, push, auth, security, observability, release. Chapters 1 and 27 are about the role. The rest is the system.

Trade-offs, priced

How long should an access token live? The spec sets no number. The book compares what Okta, Microsoft Entra and Auth0 default to (an hour, 60 to 90 minutes, 24 hours), then prices the choice in refresh traffic and battery. It argues from RFCs, vendor documentation and engineering posts, and cites them.

Failures, in order

Most chapters include a failure timeline. Chapter 22 runs an emergency signing-key rotation while half the fleet is offline and shows what each user saw. Others cover a screen that retried a “no” (chapter 11), a new status value meeting ten million strict decoders (chapter 13), the resurrected task (chapter 17) and a green canary with a doubled crash rate (chapter 26).

One method, used throughout

Chapter 4 gives you a six-step framework you can run live in an interview or an architecture review, and the Mobile Systems Assurance Map, a fourteen-field trace from the user’s promise to the signal that proves it holds. From chapter 6 on, every chapter fills in one row of it.

FROM THE BOOK

The depth behind the role

Here is what that looks like on the page. Below is one section of chapter 22, authentication and authorization done right on mobile, word for word as it is in the book. It takes a control most apps ship, shows the failure a phone’s network causes in it, and gives the design choice that follows.

FROM CHAPTER 22 OF ALTITUDE

Refresh rotation, reuse detection, and the train problem

Turning on a refresh token is not enough. A refresh token that never rotates and never expires is a finding in any modern security review, because a single leaked one becomes permanent account access with no detection signal. The fix is rotation: every time the client redeems a refresh token, the server issues a new refresh token and invalidates the old one. Now a leaked token is only useful until the legitimate client refreshes next, and you get a detection signal for free.

That signal is reuse detection, and it rests on one idea: the token family. Every refresh token the server issues from a single login is a link in a chain. When the client redeems link N, the server records that N is now spent and issues link N plus one. A correct client only ever holds the newest link, so a spent link should never appear again. The moment a spent one does, the server knows two parties hold copies of the same chain, and it cannot tell which is the legitimate one, so it revokes the entire family and forces reauthentication. Auth0 documents exactly this behavior.

Make the race concrete, because the staff-level point lives in the symmetry. A user logs in and gets refresh token RT1. The client refreshes and gets RT2; RT1 is now spent. Somewhere in here an attacker has stolen a copy of RT2, off a compromised proxy, out of a backup, however. Now both the legitimate client and the attacker hold RT2. Two cases, and the system handles both the same way.

Case one: the attacker moves first. The attacker redeems RT2 and gets RT3. RT2 is now spent on the server. The legitimate client, still holding RT2, refreshes on its normal schedule, presents RT2, and the server sees a spent token. Alarm. The family, RT3 included, is revoked, the attacker is locked out, and the real user is bounced to re-authenticate.

Case two: the legitimate client moves first. The client redeems RT2 and gets RT3. RT2 is spent. The attacker, still holding RT2, redeems

it, and the server sees a spent token. Same alarm, same family revocation, same outcome: both parties lose access and the real user re-authenticates from a clean credential.

The elegance is that the server never has to guess who is the thief. Whoever presents the spent link second trips the trap, and the response is identical either way: burn the family, force a fresh login. The cost is a forced re-auth for the real user, and what that cost buys is the signal. Without rotation a stolen refresh token is silent forever; with rotation and reuse detection, the second use is a detection event you can alert on, attribute to a device and IP, and feed to your fraud pipeline.

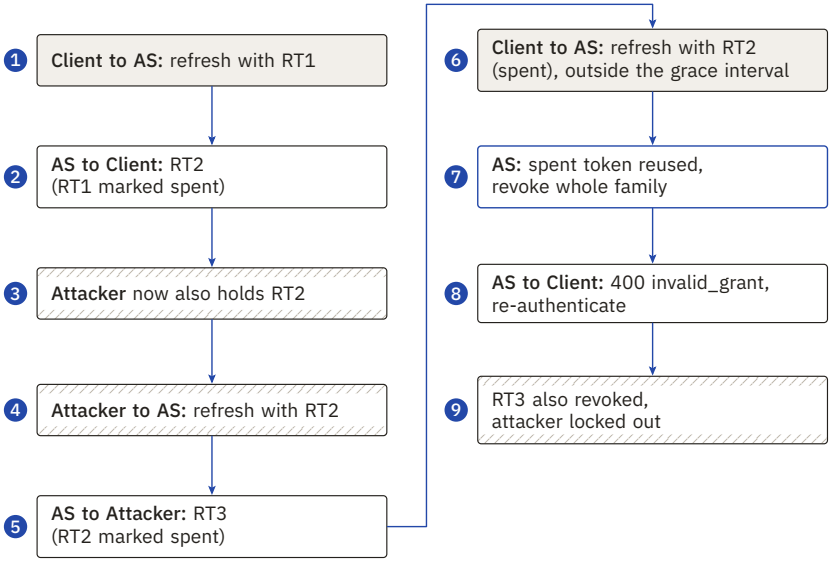


FIGURE 22.3 Refresh token rotation with reuse detection, showing how a replayed spent token trips family revocation and locks out the attacker

Beautiful in theory. Then you ship it to phones, and your support queue lights up with “I got logged out randomly.”

The train problem is the detail that separates a staff answer from a textbook one. A mobile client on a flaky network sends a refresh request, the server rotates the token and sends back the new one, and the response dies in a tunnel before it arrives. The client never

got the rotated token. So it retries with the old one. To a naive reuse-detection implementation, that retry is indistinguishable from an attack: an already-redeemed token, presented again. The system revokes the family and logs out an innocent user who did nothing but ride through bad coverage.

The fix is a deliberate usability valve. Auth0 and Okta both expose a configurable reuse interval, a small grace window during which a recently-rotated token can be presented again without tripping the alarm. Okta calls it the grace period for token rotation: 30 seconds by default, configurable from 0 to 60 (Okta Developer). Auth0's reuse interval is its own setting with its own limits, so check the number on the vendor you use rather than assuming parity. You set a grace interval of tens of seconds, so a client that sent a request, lost the response, and retried within the window gets a working token rather than being punished for the network's sins. The valve has a price: an attacker who replays the spent token inside the same window also gets tokens and trips no alarm, so the second-use guarantee above holds only outside the window. Within the window these vendors issue tokens again; they do not promise to hand back the same pair. If you run your own authorization server, you can go further and make the refresh idempotent, returning the previously issued pair for a retried request; that is a design you build yourself, because no server gives it to you by default.

This generalizes past auth: an obvious security control often needs a deliberate usability valve, because the network is hostile in ways the spec authors did not have to live with.

MORE IN CHAPTER 22

The chapter goes on to token storage in the Keychain and the Keystore, sender-constrained tokens, passkeys, and a failure timeline: an emergency key rotation while half the fleet is offline. It ends with interview questions, exercises and a worked solution.

Get Altitude on Gumroad

Where you practise

The Companion is where you practise. This is Worksheet 22 from the Workbook, the same design problem as the section above, without its evaluation criteria and working space. Two lines from its review notes in Volume 04 follow.

WORKSHEET 22 · BUILD

Set the token lifecycle for a messaging app

OBJECTIVE

Choose three numbers and a storage class, justify each against the threat model, and make a stolen-phone logout real.

SCENARIO

A consumer messaging app with hundreds of millions of users. No money moves, but conversations are private. A notification service extension decrypts messages while the phone is locked.

ASSUMPTIONS AND CONSTRAINTS

- The access token is a stateless JWT.
- Use the vendor defaults the book compares; check current documentation before relying on them.

INSTRUCTIONS

1. Choose the access-token lifetime and say where it sits in the vendor spread and what it costs in refresh traffic and battery.
2. Choose the refresh rotation policy and reuse-detection grace interval, and say who decides that interval.
3. Choose the storage class on iOS and Android, accounting for the extension.
4. Describe logging out a user within seconds after a stolen-phone report.

DELIVERABLES

A table of the three choices with justifications, and one paragraph on revocation.

FROM THE REVIEW NOTES, VOLUME 04

Disqualifiers. “Revoke the JWT”. A universal grace number. Claiming hardware storage makes the token unusable to malware.

Push further. The extension and the app both refresh. How do you stop one of them presenting a refresh token the other has already rotated?

What the full book covers

I **Foundations**, Chapters 1 to 5
The role, the client as a node, the design framework, domain boundaries

II **Client architecture**, Chapters 6 to 10
Modules, server-driven UI, super apps, performance, governance

III **Contracts and data**, Chapters 11 to 20
API and BFF design, API evolution, caching, offline-first, sync and conflict resolution, distributed systems, resilience, events

IV **Real time, trust and operations**, Chapters 21 to 26
Real-time delivery and push, auth, security, observability, the delivery system, cloud and Kubernetes

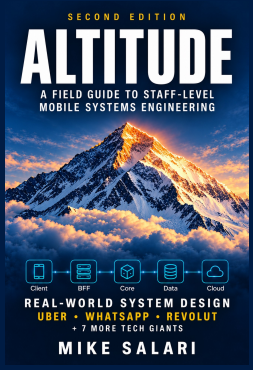
V **The role and the cases**, Chapters 27 to 30
Becoming the staff mobile engineer, then three case studies

The book

- 556 pages, 30 chapters, 5 parts. Second edition, PDF and EPUB.
- Three case studies, 91 pages: ride-hailing, global messaging, high-assurance financial. Each runs the same 13 steps, from user promises to unresolved trade-offs, with an injected incident to contain and recover from.
- Every chapter: interview questions with what a strong answer covers, and warm-up, build and stretch exercises with one worked solution printed.

The Engineering Companion

- 4 PDF volumes, 410 pages. Workbook: 30 worksheets (one per chapter), 6 game-day scripts and an interview bank for every chapter.
- Field Kit: 16 templates, 5 filled examples and 6 pre-review checklists. Guide: a staff competency rubric and 3 capstones. Worked Solutions: review notes and the chapter solutions the book does not print.



GET THE FULL BOOK

Altitude, second edition

You came for what the staff job asks. Altitude is how you prepare for the technical half: the designs, the trade-offs, the failures, and the interview that tests them.

556 pages, 30 chapters, three case studies, interview questions and exercises in every chapter, and a 410-page Engineering Companion to practise on.

Get Altitude on Gumroad

Read more at salari.dev

ABOUT THE AUTHOR

Mike Salari is a Staff mobile engineer with 15+ years across Apple, Adobe, Cisco, Visa, Mastercard and Deloitte, and 500+ technical interviews from the hiring side. He writes about software engineering and careers at salari.dev.

salari.dev LinkedIn

© 2026 MIKE SALARI. ALL RIGHTS RESERVED. SHARE THE LINK FREELY.