THE iOS ENGINEER PLAYBOOK

Get Found, Ace Interviews,
Negotiate Better Offers,
and Grow Your Career

MIKE SALARI

13+ Years Experience

Tech Lead • Architect • Hiring Interviewer

The iOS Engineer Playbook

Get Found, Ace Interviews, Negotiate Better Offers, and Grow Your Career

Mike Salari

Senior iOS Engineer · Tech Lead · Architect · Interviewer

13+ years across enterprise, payments, creative tools, and consulting

2026 Edition

Salari Press

Table of Contents

State, lifecycles, layout, performance, and choosing between them.

- **Chapter 12 - Architecture**

MVC, MVVM, VIPER, Clean Architecture, modularity, and how to choose.

- **Chapter 13 - Networking**

URLSession, REST, Codable, caching, errors, and resilience.

- **Chapter 14 - Persistence**

UserDefaults, Keychain, files, Core Data, and SwiftData.

- **Chapter 15 - Concurrency**

GCD, operations, async/await, actors, Sendable, and data races.

- **Chapter 16 - Testing**

Unit, integration, UI tests, TDD, mocking, and test strategy.

- **Chapter 17 - Live Coding**

The complete playbook for the moment everyone fears.

- **Chapter 18 - System Design**

Designing real apps: feeds, chat, offline-first, photo apps.

Part IV - Behavioral & Leadership

- **Chapter 19 - Behavioral Interviews**

Building a story bank and proving ownership, judgment, and growth.

- **Chapter 20 - Leadership Interviews**

Influence, direction, and the senior-vs-staff expectations that get tested.

- **Chapter 21 - Stakeholder Management**

Working with PMs, design, and leadership; communicating up and across.

- **Chapter 22 - Conflict Resolution**

Disagreement handled with maturity - and what interviewers listen for.

- **Chapter 23 - Mentoring & Technical Influence**

Growing others, raising the bar, and impact through other people.

Part V - Offers & Career Growth

- **Chapter 24 - Compensation Negotiation**

Total comp, market data, and negotiating salary and equity without fear.

- **Chapter 25 - Contracting vs. Employment**

Rates, trade-offs, and choosing between contract and full-time work.

- **Chapter 26 - Evaluating Companies**

Startup vs. enterprise, remote, growth, managers, and reading the red flags.

- **Chapter 27 - Accepting the Right Offer**

Comparing competing offers, deciding well, and resigning cleanly.

- **Chapter 28 - First 90 Days**

Onboarding that confirms the hire and sets up your next promotion.

Appendices

- Appendix A - Quick Revision Cheat Sheets
- Appendix B - The 30-Day Interview Prep Plan

Back Matter

- Resources & Credits
- Legal Notes

Copyright

First edition: 2026.

Published by Salari Press.

For permissions, bulk orders, or corporate training inquiries: `mike@salari.dev`

About the Author

Mike Salari, Senior iOS Engineer, Tech Lead, Architect & Interviewer

Mike Salari is a Senior iOS Engineer, tech lead, and architect with more than **13 years** of experience designing, building, and shipping iOS applications used by millions of people.

Over his career he has worked at **Cisco, Mastercard, Adobe, Deloitte, and Endava** - across enterprise platforms, global payments, creative tools, and consulting. He has worn nearly every hat on an iOS team: the junior fixing his first bug, the mid-level engineer owning a feature end to end, the senior owning a payments SDK, the architect drawing the boxes and arrows, and the tech lead and interviewer sitting on the *other* side of the table.

That last part matters for this book. As a tech lead and hiring interviewer, Mike has **interviewed and helped hire dozens of iOS engineers** at every level, from brand-new graduates to staff engineers and tech leads. He has written the rubrics, calibrated the scorecards, debated candidates in hiring committees, screened resumes by the hundred, negotiated offers, and delivered both the "we'd love to have you" and the "not this time" calls.

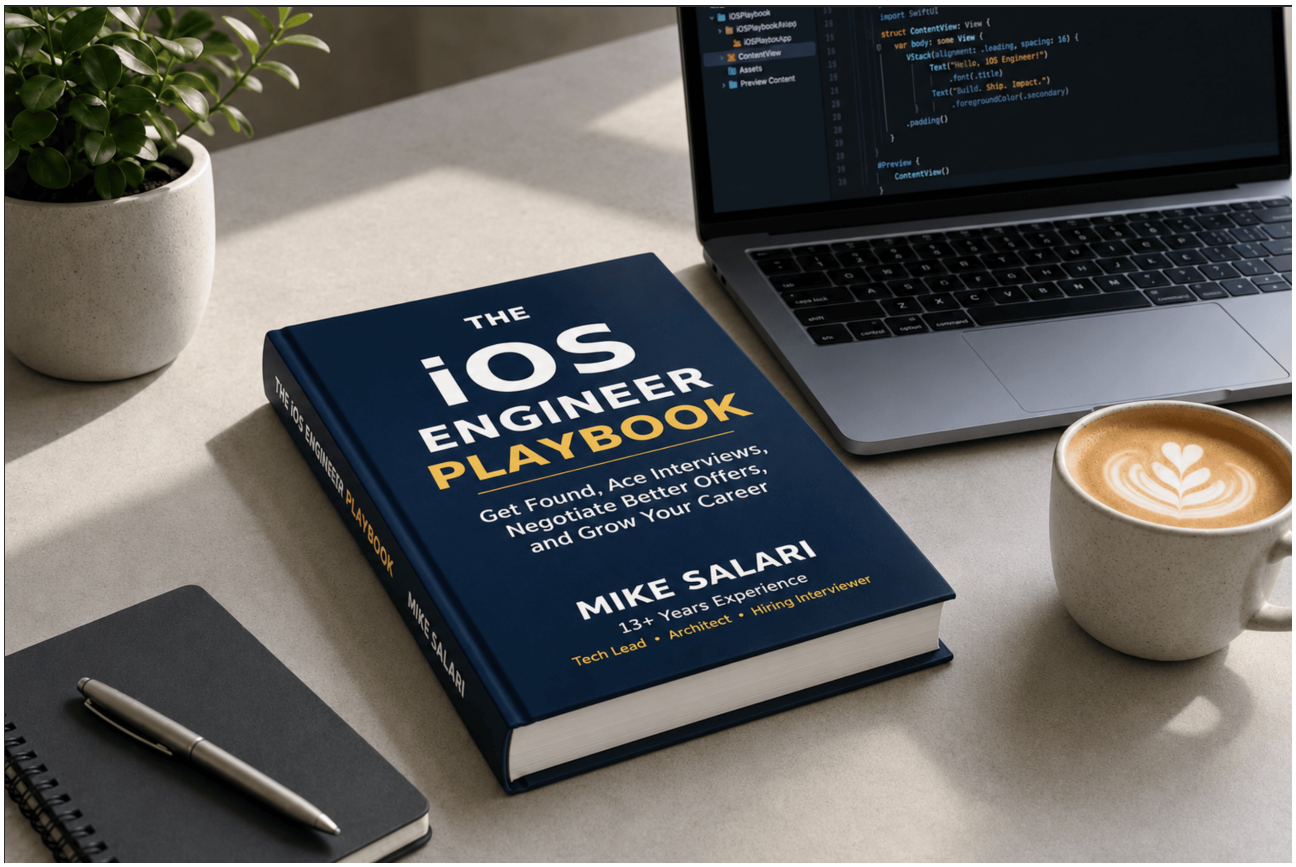
This book is built around that perspective: not just what a candidate should *do*, but what an interviewer is trying to prove, what a scorecard needs to support, what a recruiter is screening for, and what a hiring committee can confidently defend.

> "Most people don't lose iOS jobs because they lack talent. They lose them because nobody ever showed them how the hiring system actually works - how to get found, how to be evaluated, and how to turn an offer into the right offer. My goal is to hand you the whole map, end to end." - Mike Salari

Connect:

- Website: `salari.dev`
- LinkedIn: `linkedin.com/in/mike-salari`
- GitHub: `github.com/salari-dev`
- Email: `mike@salari.dev`

How to Use This Book



Your companion for every stage of the iOS hiring journey

Welcome. I'm genuinely glad you're here.

Most books about getting an iOS job are one of three things: a Swift tutorial, a bank of interview questions, or a pile of generic career motivation. This is none of those. **This is a roadmap** - a single, ordered path that takes you from *invisible candidate* to *signed offer*, and then into the first months of the job.

Every chapter has one job: to move you one concrete step closer to getting hired. If a chapter wasn't going to change what you *do* on Monday, I cut it.

Who this book is for

This book is for working iOS engineers at every level who want to get hired - and hired at the right level:

- **Junior engineers** trying to get their first real iOS role and stand out

without years of experience.

- **Mid-level engineers** who keep hearing "we went with someone more senior" and need to understand exactly what that means - and how to close the gap.
- **Senior engineers** who must nail architecture trade-offs, system design, and the leadership stories that separate a "hire" from a "strong hire."
- **Staff engineers and tech leads** stepping into ambiguous, scope-defining loops where the bar is influence and judgment, not syntax.

You do **not** need to read it cover to cover. But it is *designed* to reward those who do, because hiring is a sequence: getting found enables getting interviewed, which enables getting an offer, which enables negotiating it well.

How the book is organized

The book follows the real arc of getting hired, in five parts:

- **Part I - Getting Found.** Become visible to the right people. Resume, LinkedIn, GitHub, portfolio, personal brand, recruiters, networking, and referrals.
- **Part II - Preparing for Interviews.** Understand how hiring teams think, how leveling works, and how to answer questions with structure and confidence.
- **Part III - Technical Interviews.** The depth that earns the offer: Swift, UI, architecture, networking, persistence, concurrency, testing, live coding, and system design.
- **Part IV - Behavioral & Leadership.** Behavioral and leadership rounds, plus stakeholder management, conflict, mentoring, and technical influence.
- **Part V - Offers & Career Growth.** Compensation negotiation, contracting vs. employment, evaluating companies, accepting the right offer, and the first 90 days.

The leveling system

Throughout the book you'll see four level tags. They map to how companies actually think about scope and seniority:

- **[Junior]** - You can execute well-defined tasks with guidance.

- **[Mid]** - You own features end to end.
- **[Senior]** - You own systems and make sound trade-offs.
- **[Staff/Lead]** - You own architecture, direction, and the people around you.

Where advice differs by level, I say so explicitly. When you read a question, **find your level first**, master that answer, then read one level up. That "one level up" reading is the single fastest way to get promoted in an interview - and hiring committees use the *whole* ladder to calibrate you, so the contrast is the point.

How each chapter is structured

This is a roadmap, so every chapter is built the same way. You always know where you are and what to do next. Each chapter contains:

1. **Why this matters** - the stakes, and what changes if you get this right.
2. **Real-world examples** - real scenarios, real code, real mistakes, real

outcomes - not theory.

1. **Common mistakes** - the patterns that quietly sink candidates.
2. **Interviewer insight** - what's happening on the *other* side of the table while

you answer.

1. **Action checklist** - concrete steps to take this week.
2. **Key takeaways** - the distilled summary for quick revision.

Technical chapters add a leveled progression - fundamentals, intermediate, senior - plus common interview questions, model answers, and red flags.

Notes from the other side of the table

Scattered through this book you'll find a special kind of box - labeled **From the other side of the table**, **In the debrief**, **On the scorecard**, **What the interviewer is thinking**, or **Hire or strong hire**. These are the parts most books leave out.

I've spent years on hiring panels: screening resumes, writing scorecards, sitting in debriefs, and arguing for and against candidates in calibration meetings. Those boxes are where I tell you what's happening on *my* side of the table - what I'm quietly writing down, what raises my confidence, what plants doubt, and why two engineers with nearly identical skill walk out with different decisions.

A promise and a request

My promise: every piece of advice here is something I'd act on myself, or accept as a real interviewer - and I'll always tell you *why* it works.

My request: don't memorize - *understand*. Interviewers and recruiters can smell a script from across the room, but they lean in when they hear genuine understanding.

> **Companion resource:** This book covers the *whole* hiring journey - visibility, resume, LinkedIn, recruiters, interviews, negotiation, and career growth. If you want to go deeper specifically on technical interview preparation, a companion volume, *iOS Interview Blueprint*, focuses entirely on that - more Swift, architecture, networking, and concurrency practice with model answers. Use this book for the full path, and reach for that one when you want extra technical drilling: https://mikesalari.gumroad.com/l/ios-interview-blueprint

Now, let's begin where every hire begins: getting found.

Part I - Getting Found

The best technical skills in the world are worthless if no one ever sees them. Most qualified iOS engineers don't lose offers in the interview - they lose them long before, by being invisible. Their resume never clears the filter. Their LinkedIn never surfaces in a recruiter's search. Their GitHub looks abandoned. No one referred them.

This part fixes that. It turns you from a candidate who *applies and waits* into a candidate that opportunities come *to*. We'll build the assets that work for you 24/7 - resume, LinkedIn, GitHub, portfolio, and personal brand - and then the human network that quietly drives most senior hires: recruiters, referrals, and relationships.

By the end of Part I, you'll have a system that generates interviews. Everything after this part is about converting them.

Chapter 1 - Understanding the iOS Hiring Market

Why this matters

Before you fix your resume, before you study a single Swift concept, you need a map of the territory. Most candidates skip this step, and it costs them. They optimize the wrong things, apply through the lowest-converting channel, and walk into a process whose shape they don't understand - which makes every stage feel like a surprise. Surprises create anxiety, and anxiety makes you perform below your real level.

This chapter gives you the map: how iOS roles actually open, where companies find the people they hire, the predictable shape of the interview loop, and the current market realities you're operating in. By the end you'll know the *game* you're playing - and the rest of the book teaches you how to win each stage of it.

Here's the single most important idea to start with:

> **Mental model:** Getting hired is a funnel, and most candidates leak at the very top. The hardest part for most qualified engineers is not passing interviews - it's *getting into one*. We attack the funnel in order: get found (Part I), get evaluated well (Parts II-IV), get the right offer (Part V).

First principles: what is a company actually buying?

Start with the hiring manager's real concern.

When a company opens an iOS role, they are not "hiring a person who knows Swift." They are trying to **reduce a risk**. They have work that needs doing - features to ship, systems to maintain, fires to prevent - and every hire is a bet that *this specific person will make the team better and the product stronger than they cost.*

So every step of the process, no matter how technical it sounds, is secretly probing one of three questions:

1. **Can you do the work?** (Technical competence)
2. **Will you do the work well, with us?** (Collaboration, communication, judgment)
3. **Are you a risk we can afford?** (Reliability, growth, culture, ego)

Hold onto these three questions. Every chapter in this book is, ultimately, about helping a company answer *yes, yes, and no* with confidence. Get found, and you get the chance to answer them. Answer them well, and you get the offer.

How iOS roles actually open

Roles don't appear on a job board out of nowhere. Understanding the moment a role is born tells you when and how to be visible.

- **Backfill.** Someone left. The team is now short-handed and a little stressed,

and they want to hire *fast*. These roles move quickly and reward candidates who are already visible and easy to evaluate.

- **Growth headcount.** The team got budget to expand. These are less urgent, more

selective, and often more open to "raising the bar" - which favors strong, well-leveled candidates.

- **Backfill-in-disguise / quiet pre-posting.** Many roles are filled, or

half-filled, *before* they're ever publicly posted - through referrals and recruiter sourcing. By the time a hot role hits LinkedIn, an internal candidate or a referred one may already be in the pipeline.

> **Senior note:** The best roles are often the least visible. The last time I needed a senior iOS engineer for a payments team at Mastercard, the first thing I did wasn't post the req - it was ask the three engineers I trusted most, "who do you know?" Two strong candidates were in conversations before the role was ever public. That's not a conspiracy; it's risk reduction - and the more senior the role, the more it happens. Part I's whole job is to put you *inside* that trusted set.

The five paths into a company (ranked by conversion)

Every candidate enters through one of five doors. They are wildly unequal. Spend your energy proportional to conversion, not to comfort.

1. **Referral (highest conversion).** A current employee vouches for you. Your

resume skips the pile and lands on a desk with a recommendation attached. Referrals convert to interviews and offers at multiples of cold applications. (Chapter 5 is devoted to earning them.)

1. **Inbound recruiter sourcing.** A recruiter finds *you* - usually via LinkedIn or GitHub - because your profile matched their search. You start the conversation with someone already predisposed to like you. (Chapters 3-4 make you findable.)
1. **Warm network.** A former colleague, a community connection, or someone who saw your work reaches out or makes an introduction. Slower to build, but compounding.
1. **Targeted application with a hook.** You apply cold, but you've tailored the resume and you have *something* (a referral request in flight, a relevant project, a thoughtful note) that lifts you out of the pile.
1. **Cold application (lowest conversion).** Your resume hits an applicant tracking system (ATS) in a stack of hundreds. If it doesn't match the role's keywords, a human may never see it.

Most candidates spend 90% of their effort on path 5 - the one that converts worst - because it *feels* like progress to submit applications. Strong candidates invert that. They make themselves findable (paths 2-3), pursue referrals aggressively (path 1), and treat cold applications as a backstop, not a strategy.

> **The golden rule:** Effort should flow to the highest-converting path you can access. One earned referral is worth more than fifty cold applications - and takes less time.

The shape of the iOS interview loop

Almost every iOS interview process - at a startup, a bank, a FAANG, or a consultancy - follows the same skeleton. The names and order shift, but the stages are remarkably consistent. Knowing the whole map keeps you calm at every stage.

The iOS interview loop, stage by stage

Stage 1 - Application & sourcing

This is where most candidates lose before they begin - not because they're unqualified, but because they're invisible. (See the five paths above. Parts I is all about winning this stage.)

Stage 2 - Recruiter screen

A 20-30 minute call with a recruiter, not an engineer. They're checking: Are you real? Are you interested? Are your expectations (level, location, compensation) compatible? Can you communicate like a professional? This is **easy to pass and easy to fumble.** Be warm, concise, know why you want *this* company, and don't anchor yourself badly on compensation. (Chapters 5 and 24 cover this directly.)

Stage 3 - Technical phone screen

Now an engineer enters. Usually 45-60 minutes - a shared editor, a medium coding problem, iOS-specific questions, or a "tell me about a project" deep dive. The bar is "*does it seem plausible that this person can code and reason?*" You don't have to be perfect. You have to be clearly competent and clearly communicative.

Stage 4 - Live coding

The round everyone fears, which is why all of Chapter 17 is devoted to it. You write code, live, while someone watches. The secret most people miss: **the code is only half the score.** The other half is *how you think out loud, clarify, and recover.*

Stage 5 - The onsite loop

The main event: 3-6 back-to-back interviews (now usually virtual) covering more coding, iOS/domain deep dives (architecture, concurrency, networking, UI - Part III), system design (Chapter 18), and behavioral/leadership (Part IV, decisive at senior+). Each interviewer writes an independent score. Then they meet.

Stage 6 - Debrief, offer & negotiation

The interviewers gather (the "debrief" or "hiring committee") and argue your case against a rubric. How that room works - and how to feed it - is Chapter 6. If the signal is strong and consistent, you get an offer, and then the negotiation begins, which is its own skill (Part V).

> **What interviewers really want (at every stage):** consistent signal. One amazing answer and three confusing ones is a *no*. Four solid, clear, honest answers is a *yes*. Consistency beats brilliance.

The current market: what's actually true right now

The fundamentals of hiring don't change much. But the content and climate do, and today's market has some specific realities worth naming so you prepare for the right things.

Technically, what teams expect:

- **SwiftUI is the default**, not the experiment - while UIKit still runs an

enormous amount of production code, so interop knowledge matters (Chapter 11).

- **Swift Concurrency** (`async/await` , actors, `Sendable`) is table stakes at mid+

levels, and Swift 6 strict concurrency is a hot topic (Chapter 15).

- **SwiftData** has matured alongside Core Data; know when each fits (Chapter 14).
- **AI-assisted development** is everywhere. Interviewers increasingly ask how you

use code assistants responsibly - and many live-coding rounds now explicitly disable them, precisely to see *your* reasoning.

Climate-wise:

- The market rewards **depth and clear communication** over buzzword breadth. "I've

touched 15 frameworks" loses to "I deeply understand 5."

- **System design and behavioral rounds carry more weight** than they used to, even

for some mid-level roles, because companies hire more carefully.

- **Take-home + live review** formats are common: you build something small at home,

then walk through and extend it live. This rewards clean, explainable code.

- **Hiring bars rose and headcount tightened** after the boom years. Fewer roles,

more candidates per role, and more scrutiny per hire. That makes Part I (getting found and standing out) more important than ever - the pile is deeper.

> **Senior note:** Don't chase every new API to look current. Strong interviewers are far more impressed by someone who can explain *why* a `struct` is a value type and what that means for concurrency than by someone who name-drops the newest beta framework. Depth signals seniority; breadth signals nervousness.

Real-world example: reading a job description like an interviewer

Here's a skill that pays off immediately. A job description is a **rubric in disguise** - and a map of which rounds are coming. Learn to decode it.

Consider this typical posting:

```
Senior iOS Engineer

- 5+ years building and shipping iOS apps in Swift
- Strong understanding of SwiftUI and UIKit
- Experience with modern concurrency (async/await)
- Experience designing scalable, testable architecture
- Mentor junior engineers and drive technical decisions
- Nice to have: experience with CI/CD and performance profiling
```

Now decode it the way the hiring manager who wrote it would:

- **"5+ years... shipping"** → They'll probe whether you've owned things *in production*, not just built toy apps. Prepare shipping stories with outcomes.
- **"SwiftUI *and* UIKit"** → Expect a question on interop or "when would you still reach for UIKit?" (Chapter 11).
- **"modern concurrency"** → An `async/await` /actor question is almost certainly coming (Chapter 15). Must-study, not nice-to-have.
- **"scalable, testable architecture"** → A system design and/or architecture round is near-certain (Chapters 12 & 18). They care about *testability* specifically - have an injection/mocking story ready.
- **"Mentor... drive technical decisions"** → A *leveling signal*: they expect senior behavioral evidence (Part IV). Mentorship and influence stories are not optional here.
- **"Nice to have: CI/CD, profiling"** → Bonus points, not blockers. Mention if you have it; don't panic if you don't.

> **Senior note:** Mapping the JD to likely rounds *before* you prep means you study the *right* things. Two candidates with equal skill but unequal targeting get very different outcomes. Spend 15 minutes on this for every role you take seriously.

How the market differs by level

The same market treats a junior and a staff candidate very differently. Knowing where you sit changes your whole strategy.

- **[Junior]** The hardest part is *credibility* - you're competing against many applicants and have little track record. Your leverage is demonstrated work (projects, GitHub) and coachability. Cold applications convert worst for you, so referrals and a strong portfolio matter most. Volume of *quality* applications, plus visible proof you can ship, is the game.
- **[Mid]** You have a track record but face the most crowded band. Differentiation

comes from clear ownership stories and avoiding the "down-level" (Chapter 7). Inbound starts to work as your profile strengthens.

- **[Senior]** Inbound and referral dominate. Recruiters search for you. Your brand and network do real work. The bar shifts from "can you code?" to "can we trust your judgment?" - so system design and leadership signal carry the offer.
- **[Staff/Lead]** Almost entirely network-driven. Roles are often scoped *around* a person. Your reputation, writing, talks, and the people who vouch for you are the market. Public job postings are the least relevant path you have.

Common mistakes

- **Spending all your effort on cold applications** because submitting *feels* productive. It's the worst-converting path. Invert it.
- **Starting your prep with LeetCode** before your resume and LinkedIn can even get you into a room. Fix the funnel top-down.
- **Treating the recruiter screen as a formality.** It's a real gate, and recruiters become your advocate - or your blocker.

Interviewer insight

> **From the other side of the table:** By the time a role is posted, I've usually already asked my team "who do you know?" The first names that come back through referral get reviewed first, with the benefit of the doubt, and against a softer filter than the cold pile. I'm not being unfair - I'm reducing risk with information I trust. The lesson for you isn't to resent that; it's to *be* one of those names. Everything in Part I is about getting your name into that early, trusted set before the pile even forms.

> **On the scorecard:** Long before any scorecard exists, there's a quieter filter: the ten-second resume scan and the LinkedIn click. I'm not reading deeply - I'm hunting for a reason to say "let's talk." A resume that generates a *question I want to ask you* earns the call. A resume that generates nothing earns a quiet pass. You control that first impression completely, and most candidates never optimize it.

Key takeaways

- A company isn't buying "someone who knows Swift" - it's **reducing risk**. Every stage probes: *Can you do the work? Will you do it well with us? Are you a safe bet?*
- Getting hired is a **funnel**, and most qualified engineers leak at the top by being invisible. Attack it in order: get found, get evaluated well, get the right offer.
- The five paths in are wildly unequal. **Referral > inbound > warm network > targeted application > cold application.** Spend effort proportional to conversion, not comfort.
- The loop is predictable: **application → recruiter → tech screen → live coding → onsite loop → debrief & offer.** Knowing the whole map keeps you calm at each stage.

Chapter 2 - Resume Optimization

A resume is a marketing document, not an autobiography

Why this matters

When I screened resumes for iOS openings at Deloitte, I'd move through a stack of forty in under ten minutes - each one got me for a few **seconds** before I decided to keep reading or move on. And before a human like me ever sees it, an algorithm may decide whether it surfaces at all. This is the highest-leverage document in your entire search: the best Swift skills in the world are worthless if your resume never gets you into the room.

Here's the mindset shift that changes everything: **your resume has exactly one job - to get you the interview.** It is not your autobiography, not a complete list of everything you've touched, and not the place to be humble. It is a focused, one-to-two-page marketing document whose only goal is to make a busy person think: *I should talk to this person.*

Two audiences read it, in this order, and you must satisfy both:

1. **The ATS (Applicant Tracking System)** - software that parses, stores, and ranks resumes, often filtering by keywords before a human looks.
1. **The human** - a recruiter, then a hiring manager, who skims for relevance and signal.

Let's build a resume that wins both.

Beating the ATS (without keyword stuffing)

An ATS reads your resume as plain text and tries to extract structured data: your name, jobs, skills, dates. If it can't parse your fancy two-column design, your information gets garbled or dropped, and you score poorly against the role's keywords - so you never reach a human.

Rules that keep the ATS happy:

- Use a **single-column layout.** Multi-column resumes and text inside graphics often parse into nonsense.
- Use a **standard, readable font** and real text - never put important information inside an image, icon, or header/footer region.
- Use **standard section headings**: "Experience," "Skills," "Education," "Projects." Don't get clever with "Where I've Made Magic."
- Submit a **PDF** unless the application explicitly asks for `.docx`. Most modern systems handle PDF well; a few legacy ones prefer Word.
- **Mirror the job's language.** If the posting says "Swift, SwiftUI, REST APIs, unit testing," and you've done those, use those *exact* words. The ATS matches literal terms; "wrote tests" may not match "unit testing."

> **What interviewers really want:** keyword *alignment*, not keyword *stuffing*. Listing "Swift" fifteen times in white text is a 1990s trick that modern systems flag and humans find dishonest. Use the right terms naturally, in context, backed by real accomplishments.

Achievements, not duties (the #1 fix)

This single principle separates forgettable resumes from interview-winning ones. Most people write **duties** - what they were *supposed* to do. Strong candidates write **achievements** - what they actually *accomplished*, with a result.

A simple, powerful formula:

> **Accomplished [X], as measured by [Y], by doing [Z].**

Watch the transformation:

- Weak (duty): "*Responsible for the app's networking layer.*"
- Better (action): "*Built the app's networking layer using URLSession and Codable.*"
- Strong (achievement): *"Rebuilt the networking layer with URLSession and

async/await, cutting average screen-load time by 38% and crash rate by 22%."*

The third version proves *impact*. Even when you don't have a perfect metric, you can almost always find a proxy: time saved, users affected, crashes reduced, ratings improved, releases unblocked, teammates onboarded.

> **Red flag (from the resume screener's chair):** a wall of "Responsible for..." bullets. It reads as someone who *occupied* a role rather than *moved* it. Lead every bullet with a strong past-tense verb (Built, Shipped, Led, Reduced, Designed, Migrated) and end with a result.

The anatomy of a great iOS resume

Here's the structure I recommend, top to bottom.

1. Header. Name, one-line title, and contact: phone, professional email, and clickable *links* to LinkedIn, GitHub, and portfolio. City and country is plenty - no full mailing address needed.

2. Summary (3 lines, optional but powerful for senior candidates). A tight pitch: who you are, your strongest area, and what you're looking for.

> "*Senior iOS Engineer with 8+ years shipping consumer apps in Swift and SwiftUI to millions of users. Specialize in performance, clean architecture, and mentoring. Looking to own the iOS platform at a product-led company.*"

3. Skills. A scannable, grouped list, so it's easy to read and easy for the ATS to match:

```
Languages:     Swift, Objective-C, SQL
iOS:           SwiftUI, UIKit, Combine, Swift Concurrency, Core Data, URLSession
Architecture:  MVVM, Clean Architecture, VIPER, Dependency Injection
Tooling:       Xcode, Instruments, Git, Fastlane, GitHub Actions (CI/CD)
Testing:       XCTest, Swift Testing, XCUITest, snapshot testing
```

4. Experience. The heart of the resume. One entry per role, reverse-chronological, each with a few **achievement bullets** (not duty bullets).

5. Projects. Especially important for juniors and career switchers. Two or three real apps or libraries, each with a one-line description and a link.

6. Education & certifications. Brief. Degree, school, year. New grads can put this higher.

The level test: tasks vs. outcomes

Before a hiring manager reads the details, your resume must pass one question:

> Does this person look like they can own outcomes, or only execute tickets?

The answer is in your verbs and your scope - and the right answer *changes by level*.

- **[Junior]** It's fine for your resume to be mostly about tasks done well, plus

projects that prove you can ship. Lead with verbs like *built, implemented, fixed, shipped, learned*. A strong portfolio project carries enormous weight here.

- **[Mid]** Show feature ownership end to end. *Owned, delivered, improved, reduced.*

Each role should have at least one bullet with a real metric.

- **[Senior]** Tasks are not enough. The verbs should show ownership of systems:

led, designed, migrated, stabilized, simplified, mentored. Weak vs. strong:

Weak senior signal:

- Implemented screens using SwiftUI.
- Worked on networking and bug fixes.
- Participated in code reviews.

Strong senior signal:

- Led a SwiftUI migration of the checkout flow, reducing UI defects and simplifying release validation.
- Reworked the networking layer around typed errors and a retry policy, cutting duplicated error handling across feature teams.
- Mentored two mid-level engineers to full feature ownership and raised code-review standards.

The difference isn't language polish. It's **scope**.

- **[Staff/Lead]** Show org-level impact and judgment: cross-team initiatives, platform work, technical direction, and *what you chose not to do*. Cut tactical bullets that read as below your level.

Tailoring without rewriting everything

You don't need a brand-new resume per application, but you should **retarget** the top third for each role. Keep a "master" resume with everything, then for each application:

- Adjust the summary line to echo the role.
- Reorder skills so the most relevant ones lead.
- Promote the two or three most relevant achievement bullets to the top of each job.

Fifteen minutes of tailoring meaningfully raises your response rate, because both the ATS and the human see *immediate* relevance.

Real-world example: a before/after that earns the interview

Let's make the achievement principle concrete with one role's bullets.

Before (duties):

```
iOS Developer, Acme (2022-2024)
- Worked on the company's iOS app.
- Responsible for bug fixes and new features.
- Used Swift and UIKit.
- Participated in code reviews.
```

After (achievements):

```
iOS Engineer, Acme (2022-2024)
- Shipped a SwiftUI redesign of the home feed, lifting App Store rating 4.1 -> 4.6.
- Cut crash rate 22% by replacing a fragile callback-based networking layer with
  async/await.
- Reduced cold-start time from 2.4s to 1.3s by deferring non-critical work off the
  launch path.
- Mentored 2 junior engineers and introduced a PR checklist that cut review cycles
  by ~30%.
```

Same person, same job. The "after" version gets the interview, because every line proves *impact* and hints at *seniority*. Notice it also naturally surfaces strong keywords (SwiftUI, async/await) for the ATS - achievements and keyword alignment working together.

> **From the other side of the table:** When a resume reaches me as the hiring manager, I'm not really *reading* it - I'm scanning for a reason to say yes in about ten seconds. My eyes jump to the most recent role and the verbs. "Responsible for the iOS app" tells me nothing, and I move on. "Cut crash rate 22% by replacing the networking layer" makes me stop, because now I have a *question I want to ask you* - and a resume that generates questions is a resume that earns an interview. If nothing on the page makes me curious, there's nothing to talk about, and that's usually a quiet pass.

Handling the hard cases

Employment gaps. A gap is fine and common. Don't hide it awkwardly or pad dates (dishonesty is far worse than a gap). State it plainly if asked, and be ready with a calm, forward-looking sentence: "*I took eight months for [caregiving / health / layoff / focused learning]; during it I [stayed sharp / built a project / took a course], and I'm fully ready now.*" Confidence is the whole game here.

Career switchers and juniors with no professional iOS experience. Lead with a **Projects** section right under your summary. A shipped app that handles real-world states (errors, offline, empty) and has tests is your strongest evidence. Frame non-iOS experience for transferable signal (ownership, shipping, collaboration).

Contract / consulting history. Group short contracts under one heading ("Independent iOS Consultant, 2021-2024") with sub-bullets per notable engagement, so a string of short stints doesn't read as job-hopping. (More on the contract path in Chapter 25.)

Too many short stints. Emphasize impact-per-stint and, where true, note the *reason* neutrally (contract ended, acquisition, relocation). Never editorialize.

Common mistakes

- **Too long.** One page for under ~8 years; two pages max for senior/staff. Length doesn't impress anyone.
- **Vague metrics or none at all.** "Improved performance" begs "by how much?"
- **Listing every technology you've ever opened once.** It dilutes your real strengths and invites brutal questions on things you don't actually know.

> **Senior note:** As you get more senior, your resume should get *more selective*, not more crowded. A staff engineer's resume that still lists "HTML, CSS, jQuery" from 2012 signals poor judgment about what matters. Cut ruthlessly. Every line should earn its place by supporting the level you're targeting.

Interviewer insight

> **In the debrief:** Over-claiming is the gift that keeps on taking. If your resume lists ten frameworks, an interviewer will pick the one *you* seem least sure about and dig - not to be cruel, but because the edge of your knowledge is where your real level lives. I've watched a candidate list "Core Data, Combine, Metal, ARKit," then crumble on the two they'd only touched in a tutorial - which cast doubt on the two they genuinely knew. List what you can defend in depth. A short, honest resume that holds up beats a long one that springs leaks.

> **If I heard a number on your resume, I'd follow up:** with "How exactly did you measure the 40%?" - and I ask that about almost every metric. It's one of the cheapest tells in the whole interview. Candidates who did the work answer instantly: the tool, the before/after, the device. Candidates who inherited or inflated the number get vague. I don't punish a humble "I didn't own the measurement, but here's the part I did." I *do* quietly downgrade a confident number that evaporates under one question. Only put numbers on your resume you can defend.

Key takeaways

- Your resume has **one job: get the interview.** It's a marketing document, not an autobiography.
- Write for **two readers**: the ATS (parseable layout, exact keywords) and the human (achievements and results).
- Use **"Accomplished X, as measured by Y, by doing Z."** Achievements beat duties every time.
- The **level test** - tasks vs. outcomes - determines whether you read as junior, mid, senior, or staff. Match your verbs and scope to your target level.