ELITE EDITION

THE

iOS INTERVIEW BLUEPRINT

Every level, from junior to engineering manager

MIKE SALARI

SECOND EDITION · 2026
Sample edition

The iOS Interview Blueprint

Elite Edition (2026)

Second edition, revised August 8, 2026

Master Your Career Trajectory, The Definitive, Career-Leveled Guide for iOS Engineers and Leaders

Mike Salari

salari.dev · mike@salari.dev

linkedin.com/in/mike-salari

Copyright

Second edition, revised August 8, 2026

Written by Mike Salari

How to read this book

This book uses the same visual grammar for every interview question. Read the first tier to confirm the foundation, the second to learn the decision rule, and the third to see the system-level consequence.

WHAT MOST CANDIDATES SAY

The complete, correct textbook answer. This establishes the baseline an interviewer expects.

WHAT A SENIOR CANDIDATE SAYS

The distinction that changes the decision, the trade-off, and a recommendation grounded in context.

WHAT A STAFF+ CANDIDATE SAYS

The effect on architecture, teams, migration, reliability, and cost at scale.

RED FLAGS

× "I always use the same solution." Strong answers start from constraints, not slogans.

× Naming a tool without explaining the failure mode it controls.

COMMON FOLLOW-UPS

? "What would change your choice?" *(The constraints, ownership, scale, or failure impact.)*

? "How would you verify it?" *(With a focused test or measurement tied to the claim.)*

Use the follow-ups as a self-test. Cover the lighter answer, respond aloud, then compare your answer with the prompt.

Table of contents

The complete contents of the full edition. Sections marked **(in this sample)** are reproduced here in full.

- SwiftData
- Swift ecosystem
- Miscellaneous
- Chapter 2, elite recap
- Level profile: what interviewers look for **(in this sample)**
- Modular design and scalability **(in this sample)**
- The Composable Architecture
- Advanced persistence
- Security best practices
- CI/CD and automation
- Modern iOS APIs and frameworks
- SwiftUI performance and the rendering pipeline
- Bridging and advanced Swift concurrency
- Launch and runtime performance
- Swift internals and performance
- Design patterns
- Senior deep-dive drills (Swift internals, patterns, APIs)
- Chapter 3, Elite recap (table) **(in this sample)**
- Level profile: How the interview changes **(in this sample)**
- Mentorship and technical leadership
- Cross-functional collaboration
- Behavioral answers: SAR with engineering specifics
- Promotion narratives and scope framing
- Negotiating level
- Kotlin Multiplatform (KMP) for iOS engineers
- Chapter 4, Elite recap (table)
- Level profile: what interviewers look for **(in this sample)**
- System design for mobile
- Team leadership & technical direction

- Strategic roadmap planning
- AI for architecture and system design
- Advanced system design walkthroughs
- Chapter 5, Elite recap (table)
- Level profile: defining the mobile ecosystem **(in this sample)**
- Multi-platform strategy
- Modularization at scale
- App architecture governance
- On-device AI strategy
- Release architecture
- Platform team topology
- Chapter 6, Elite recap (table)
- Level profile: what a mobile EM interview actually measures **(in this sample)**
- The EM loop, round by round
- Team topology: platform team versus feature teams
- Owning the release train
- Quality as a management system
- Build and CI SLAs
- Leading seniors through platform shifts
- Hiring: designing an iOS loop that predicts performance
- Behavioral rounds: the classics, answered like a manager
- Chapter 7, Elite recap (table)
- Level profile: delivery when you cannot roll back **(in this sample)**
- The physics of mobile delivery
- Planning around Apple's calendar
- Feature flags and experimentation on mobile
- Estimation and risk for mobile projects
- Coordinating iOS and Android parity

Introduction: The future of iOS interviews is here

Welcome to the elite tier

This is the **Second Edition** of *The iOS Interview Blueprint: Elite Edition* (2026). It will evolve with the platform and hiring market; your feedback helps shape later editions.

The current iOS hiring market rewards engineers who combine **depth** (Swift, UIKit, SwiftUI, concurrency, security) with judgment (architecture trade-offs, collaboration, leadership signals). Companies no longer stop at "Can you reverse a linked list?" They ask whether you can ship reliable features, explain failures, mentor peers, and design systems that survive App Store review and real-world scale.

This blueprint is **career-leveled**: each chapter maps to how interview loops actually run for junior, mid, senior, tech lead, architect, and leadership tracks. Use the chapter that matches the role you are pursuing, and skim adjacent levels to see what interviewers will expect next.

Figure I.1 shows why preparation has to cover more than code. Each round produces a different signal, and the hiring committee combines signals the candidate never sees directly.

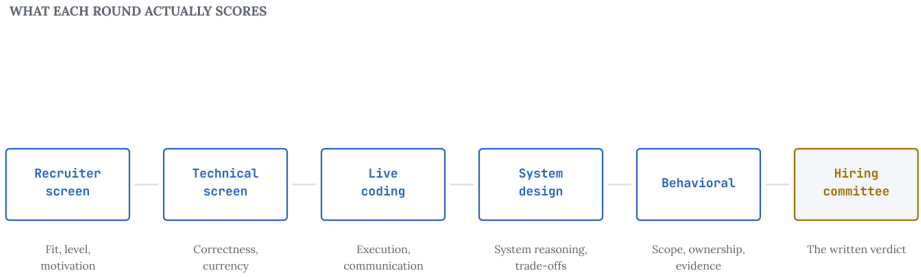

Figure I.1: Six stages, each measuring something different. The final document is the one the candidate never sees.

How the book is organized

Nine chapters, five parts:

- Part 1: Building strong foundations (chapters 1 and 2): junior and mid-level technical interviews, Swift and platform fundamentals.
- Part 2: The senior leap (chapters 3 and 4): senior-level depth, concurrency, performance, and production judgment.
- Part 3: Technical leadership (chapters 5 and 6): tech lead and architect loops, system design, and architecture trade-offs.
- Part 4: The leadership interview (chapters 7 and 8): engineering management, behavioral rounds, and working with product and project managers.
- Part 5: The hiring loop in practice (chapter 9): the live coding round, take-homes, and the modern loop, how companies actually run interviews in 2026 and how to perform in each format.

Elite tier does not mean memorizing trivia. It means:

- Answering with structure (definition □ trade-offs □ example from your experience).
- Connecting Apple platform reality (lifecycle, privacy, background limits) to design choices.

- Showing integrity in how you prepare, including thoughtful use of AI tools.

How to master this blueprint

1. Study by level, then by gap. Read your target chapter end-to-end. Each section is a question and answer, often with examples and tips. Mark topics where your answer would be weak; drill those first.

2. Speak answers aloud. Interview success is verbal. For each Q, practice a 60 to 90 second answer, then a 3-minute deep dive if the interviewer probes.

3. Build a story bank. For every mid-level and above topic, tie one production story: what broke, what you measured, what you shipped.

4. Use spaced repetition. Revisit Swift fundamentals even as a senior, loops still include ARC, value semantics, and `@MainActor`.

5. Simulate loops. Pair with a peer: one technical screen, one system design, one behavioral. Timebox.

6. Cross-check with multiple sources. This book is a foundation; Apple documentation, WWDC sessions, and your own codebase remain authoritative.

1. Track the Table of Contents. This is the master checklist, ensure you can answer every bullet for your level before onsite day.

Leveraging AI in your interview prep and beyond

Large language models are part of the modern engineer's toolkit. Used well, they **accelerate** prep; used poorly, they produce confident wrong answers about threading or App Store policy.

Do:

- Generate flashcard-style Q&A and compare to this blueprint.
- Ask for mock interviews with follow-up questions.
- Use AI to explore architectures (offline-first sync, feature modules), then validate against constraints in Chapter 5.
- Summarize WWDC sessions and draft ADR outlines for practice.

Do not:

- Rely on AI during live interviews unless explicitly allowed.
- Paste proprietary employer code into public tools.
- Ship AI-generated code without review, tests, and Instruments.

Elite mindset: You are accountable for what merges and what you say in the room. AI is a research assistant, not a substitute for judgment.

Chapter 1: Junior iOS developer (0-2 years experience)

Memory management & ARC

> **Q: How does Swift manage memory with ARC, and how do you resolve strong reference cycles?**

WHAT MOST CANDIDATES SAY

Swift uses **Automatic Reference Counting (ARC)** to manage memory for class instances. Each instance has a count of the strong references pointing at it; while that count is above zero ARC keeps it alive, and the moment it reaches zero ARC deallocates the instance immediately and deterministically, not on a garbage collector's schedule. A strong reference cycle is when two instances hold strong references to each other, so neither count ever reaches zero even after the rest of the app is done with them. That is a memory leak.

```
class Employee {
    let name: String
    var desk: Desk?
    init(name: String) { self.name = name }
    deinit { print("Employee \(name) deallocated") }
}

class Desk {
    let label: String
    var occupant: Employee?
    init(label: String) { self.label = label }
    deinit { print("Desk \(label) deallocated") }
}

var maya: Employee? = Employee(name: "Maya")
var deskA7: Desk? = Desk(label: "A7")
```

```
maya!.desk = deskA7
deskA7!.occupant = maya

maya = nil
deskA7 = nil
// Neither deinit message is printed, indicating a memory leak.
```

> *Neither* `deinit` *fires, that is the leak.*

WHAT A SENIOR CANDIDATE SAYS

The skill isn't naming `weak` versus `unowned` , it's choosing correctly from **lifetime reasoning**. Use `weak` when the referenced object can outlive you *or* die before you (delegates, parent-to-child back-references, anything you observe). The reference is optional and ARC zeroes it automatically when the target deallocates:

```
class Employee {
    let name: String
    var desk: Desk?
    init(name: String) { self.name = name }
    deinit { print("Employee \(name) deallocated") }
}

class Desk {
    let label: String
    weak var occupant: Employee?
    init(label: String) { self.label = label }
    deinit { print("Desk \(label) deallocated") }
}
// ... (rest of the example as above)
// Now, both deinit messages will be printed.
```

Use `unowned` *only* when the reference is guaranteed to outlive `self` , a child that cannot exist without its parent. It stays non-optional, so reaching for `unowned` just to avoid an optional is how you ship a crash: accessing it after the target deallocates is a runtime trap.

```swift
class Order {
    let id: Int
    var invoice: Invoice?
    init(id: Int) { self.id = id }
    deinit { print("Order \(id) deallocated") }
}

class Invoice {
    let amount: Int
    unowned let order: Order
    init(amount: Int, order: Order) {
        self.amount = amount
        self.order = order
    }
    deinit { print("Invoice for \(amount) deallocated") }
} // An invoice never exists without its order.
```

In real 2026 codebases, cycles rarely come from two objects pointing at each other, they come from escaping closures and stored `Task` values that capture `self` . Any escaping closure or retained `Task` that touches `self` needs `[weak self]` ; the classic symptom is a view model whose `deinit` never fires. Find these with the Memory Graph Debugger and Instruments (Leaks/Allocations), not by reading code, leaks hide in the paths you didn't trace.

WHAT A STAFF+ CANDIDATE SAYS

Three things mark a staff-level answer:

1. ARC has a cost. `retain` / `release` are atomic operations; in hot paths (tight loops, large arrays of reference types, per-frame work) that traffic shows up in Time Profiler. The senior fix for a leak is `[weak self]` ; the staff fix is often removing the reference type entirely, biasing to `struct` /value types so there is no ARC to manage at all.
2. Concurrency interaction. Under Swift 6 strict concurrency a captured `self` crosses isolation boundaries, so capture now interacts with `Sendable` and actor isolation, not just retain counts. `[weak self]` in a `Task` is also about whether `self` is still valid *across a suspension point*, not only about cycles.

3. Prevent it at the org level. One engineer fixing one leak doesn't scale. Install guardrails: a lint rule flagging escaping closures that strongly capture `self`, a review checklist, and a "view models must `deinit` in tests" assertion. `unowned(unsafe)` exists but should be banned outside measured hot paths.

RED FLAGS

× "Use `unowned` because it's faster or avoids the optional." Cargo-cult, it's a crash waiting on a lifetime you never verified.
× Calling ARC "garbage collection," or saying it runs "periodically."
× Reaching for `weak` / `unowned` on value types, which have no reference count.
× "I'd just read the code to find the leak." Senior engineers reach for the Memory Graph Debugger.

IF YOUR FIRST ANSWER WOBBLED

If you have already said you always use `[weak self]`, name the exception before the interviewer does: it is wrong when the work must finish regardless of the view's lifetime, an upload or an analytics flush, because a weak capture there silently drops work the user believes was saved. Saying that turns a habit into a lifetime decision, which is the thing being scored.

COMMON FOLLOW-UPS

? "You marked it `unowned`, what happens at runtime if the parent deallocates first?" (*A crash, can you defend the lifetime guarantee?*)
? "Does `[weak self]` solve every closure leak?" (*No, a strongly captured local or child object still cycles.*)
? "Structs don't use ARC, so are they free?" (*No, a struct holding a class property, or boxed in a closure or existential, still drives retain traffic.*)
? "How would you prove this code leaks in CI?" (*A `deinit` expectation test, `XCTMemoryMetric`, or the Memory Graph in a UI test.*)

> **Where candidates lose it.** Not on the definition, which almost everyone gets. On the next question, when I ask them to prove the leak. A candidate who reaches for the Memory Graph Debugger has debugged one; a candidate who offers to reread the code has only read about it.

Chapter 3: Senior iOS developer (4-7 years experience)

Micro-features and module decomposition

Figure 3.1 defines the boundary by permitted dependency direction. The Domain owns the abstractions, while infrastructure conforms from the outside.

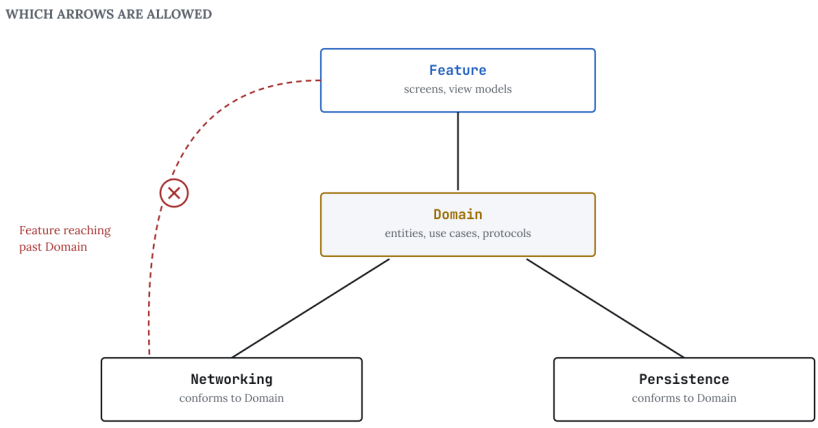

Figure 3.1: Dependencies point inward. Domain declares the protocols and the outer modules conform, which is what makes the boundary a decision rather than a belief.

Q: How do you decompose a large iOS app into modules?

WHAT MOST CANDIDATES SAY

Split the app into Swift packages by feature: a package per screen or flow, plus shared packages for networking, design system, and models. This shortens incremental build times because unchanged packages are not recompiled, and it lets teams work in parallel without merge conflicts in a monolithic target.

WHAT A SENIOR CANDIDATE SAYS

A senior candidate starts with domain boundaries, not package-count goals. A module should map to a coherent business capability with a stable, minimal public API. The single most useful structural pattern is splitting each feature into an interface module and an implementation module: `FeatureInterface` holds protocols and value types, `FeatureLive` holds the concrete implementation, and only the app target (the composition root) links implementations together. Features depend on each other's interfaces only, which breaks compile-time coupling, keeps the build graph wide and parallel, and makes SwiftUI previews and unit tests fast because they link mocks instead of the world.

Dependency direction must be enforced, not hoped for: package manifests make illegal imports impossible, and a lint rule or a build-graph check catches new edges in review. The classic failure modes are premature decomposition (thirty packages, one team), the `Shared` or `Common` dumping-ground module that quietly becomes a second monolith, and circular dependencies hidden behind convenience re-exports. After decomposition, track clean and incremental build times, test execution time per package, and defect blast radius to confirm the split paid for itself.

WHAT A STAFF+ CANDIDATE SAYS

At staff level, modularization is organizational design expressed in a build graph. You define ownership per module (codeowners, review paths, release policy for internal APIs), and you plan the migration as a strangler effort: extract leaf dependencies first, then features, and keep the app shippable at every step. You also decide the tooling question deliberately: plain SwiftPM is enough for most apps, while generated-project tools or a build system like Bazel become justified only at the scale where checked-in Xcode projects and cold CI builds are the actual bottleneck. Crucially, you keep decomposition reversible: merging two modules that turned out to be one concept is a sign of health, not failure.

Table 3.1, Decomposition quality signals

Signal	Healthy pattern
API surface	Minimal and intention-revealing
Ownership	Clear module DRI and review path
Coupling	Dependency direction enforced by manifests
Outcomes	Faster incremental builds, lower blast radius

RED FLAGS

× Splitting by technical layer only (`Networking` , `UI` , `Models`) with every feature still touching every module.
× A `Shared` module that grows without an owner or an API review.
× No measurement before and after, so nobody can say whether builds actually got faster.
× Feature modules importing each other's implementations directly.

COMMON FOLLOW-UPS

? "Feature A needs something from feature B, what is allowed?" *(A depends on B's interface module; only the app target links the implementation.)*
? "Where does a type go when two features both need it?" *(Into a domain module they both depend on, never into a* `Shared` *module with no owner.)*
? "How is the dependency direction enforced after the pull request that set it up?" *(Package manifests plus a build graph check, because convention decays.)*
? "You split into modules and the build got slower, what happened?" *(Usually a hub module everything imports, so the graph is deep rather than wide.)*
? "When would you merge two modules back together?" *(When they change together every time, which means they were one concept.)*

Level profile: what interviewers look for

Senior iOS interviews evaluate ownership depth, not just coding fluency. At this level, interviewers expect you to lead subsystem decisions, navigate trade-offs under business constraints, and keep systems healthy through change. You should demonstrate technical judgment across architecture, data, security, release operations, performance, and modern platform APIs.

The difference between a good mid-level answer and a good senior answer is rarely vocabulary. It is the ability to explain what actually happens under the hood, to connect a framework decision to its runtime consequences, and to describe how you verified a claim with a profiler, a metric, or a test instead of an opinion. Every section in this chapter is built around that standard.

Chapter 3, Elite recap (table)

Senior domain	Interview proof of mastery
Modularity	Interface/implementation splits, enforced dependency direction, measured outcomes
State architecture	Modern TCA (@Reducer, @Dependency) applied pragmatically with exhaustive TestStore coverage
Persistence	Safe multi-version migrations, CloudKit operational readiness, fixture-based CI testing
Security/compliance	Keychain access controls, key lifecycle, privacy manifest governance
Delivery systems	Risk-based CI/CD, selective testing, flake governance with ownership
SwiftUI performance	Identity, AttributeGraph dependencies, propose-choose-place layout, Instruments evidence
Concurrency	Exactly-once continuations, AsyncStream lifecycle, cancellation and priority behavior
Launch and runtime	Pre-main levers, MetricKit field data, symbolication and size discipline
Platform depth	HealthKit/ARKit/Core ML constraints, StoreKit 2 verification, modern API productization

What is in the full edition

You have just read one question worked end to end, a second at a different career level, a level profile and a chapter recap. Every block you saw appears in the full edition exactly as it appears here.

The full edition contains:

- **189 interview questions**, ordered by the career level at which they are actually asked, from junior through engineering manager and delivery.
- The three tier answer ladder on the questions that decide senior rounds, with red flags and common follow ups.
- 308 pages, checked against Swift 6.2 and the iOS 26 SDK.
- Nine chapters, including the live coding round, take homes and the modern loop.

Price: 59 USD. Get the full edition at https://mikesalari.gumroad.com/l/ios-interview-blueprint

One limit, stated plainly

This is a preparation system, not a guaranteed outcome. It cannot give you production experience you do not have. What it can do is make sure you are never surprised by a question, and never lose an offer because you answered a senior question at a mid level altitude without realising it.