FREE SAMPLE

PRACTICAL

# INTERVIEW RUNTIME

## iOS INTERVIEW PREPARATION

Swift, Concurrency, System Design, and
Mock Interviews for Senior and Staff Engineers.

MIKE SALARI

FREE SAMPLE

# The Actor That Wasn’t a Lock

*One concurrency question, answered the way a strong candidate would, then taken apart*

MIKE SALARI

From Interview Runtime 1.2: iOS Interview Preparation

---

A free sample of *Interview Runtime 1.2: iOS Interview Preparation*, 380 pages. One interview question, carried from the answer most candidates give to the one that survives the follow-up. This sample may be shared freely in its original, unmodified form.
INTERVIEWRUNTIME.COM

THE PROBLEM

# The sync that ran twice

The interviewer puts an actor on the screen and asks a question that sounds like a bug report. The shape of this prompt comes from the concurrency chapter of the book.

```
actor Syncer {
    private var isSyncing = false

    func sync() async throws {
        guard !isSyncing else { return }
        let settings = await loadSettings()
        isSyncing = true
        defer { isSyncing = false }
        try await upload(settings)
    }
}
```

> *Production logs show two complete syncs overlapping. Why?*

You have read the code for four seconds. The type is an actor, the flag is a plain stored property, and the `defer` clears it on every exit path. Here is the answer a strong senior engineer gives, and it is reasonable:

“An actor serializes access to its state, so two callers can’t touch `isSyncing` at the same time. The guard-and-set pattern should be safe. Maybe something is calling it from a detached task, or the property is `nonisolated` somewhere. Or the `defer` isn’t running when the upload throws.”

Every sentence in that answer is a thing a careful engineer would check. The first one is even true. The candidate has listed three theories, all of them about whether the flag is protected, and none of them about whether the protection covers what the code is trying to do.

That is the pattern worth noticing before the follow-up arrives. The answer is built from correct facts about actors, and it points the investig-

ation at the wrong layer. An interviewer reading it writes something like "knows the vocabulary, has not traced the code."

THE FOLLOW-UP

## One question you cannot recite your way past

The interviewer does not argue. They ask:

> *What does the `await` on `loadSettings()` do to the actor?*

The recited answer is "it suspends until the result comes back." That is accurate and it does not help, so the interviewer asks the second half:

> *While it is suspended, can anyone else run on that actor?*

This is the question the sample exists for. There is no framework name that answers it. The candidate has to hold a model of what suspension means and then apply it to this exact code, in order, out loud.

Most strong candidates feel two pulls here. One says "the actor is serialized, so the second call queues behind the first." The other, usually a beat later, says "reentrancy, right." The book's concurrency chapter treats the gap between those two as the whole subject: suspension permits reentrancy, and isolation protects memory, not logic.

Naming the word is not the test. The test is the next request, which is always some version of "walk the timeline." A candidate who says "reentrancy" and then reasons that the second call would see the flag set and bail out has the word and not the mechanism. The flag is not set yet. It is set on the line after the `await`.

The book's scorecard places this behavior in the lower rows of its first dimension: concepts defined correctly, but reentrancy and its implications missed until prompted. The red flag it lists is the sentence the first

answer was heading toward, “actors make code thread-safe” with no discussion of reentrancy.

> **WHAT THE INTERVIEWER IS SCORING**
>
> Not whether you know the word reentrancy. Whether you can trace this code, unprompted, to the line where the second caller gets in.

THE CORRECTION

## Serial entry, interleaved execution

Walk it as two callers, A and B, arriving close together.

1. A enters the actor, passes the guard, because `isSyncing` is `false`, and reaches `await loadSettings()`.
2. The `await` is a suspension point. A gives up the actor.
3. B enters, passes the same guard, because `isSyncing` is still `false`. A has not set it yet.
4. Both resume, both set the flag, both upload.

The actor did its job at every step. No memory was touched by two threads at once. The program is free of data races and wrong anyway. The invariant is “at most one sync in flight,” and the code checks it on one line and acts on it three lines later, with a suspension in between. Check-then-act across a suspension is a race condition: an ordering bug, not a memory bug.

That distinction is also the honest answer to the question an interviewer often adds: would Swift 6 strict concurrency have caught this? No. Strict checking diagnoses data races in code that opts in. This code is memory-safe throughout, so there is nothing for the compiler to report. The sentence to say is: **an actor serializes entry, not the whole method.**

### THE FIX

The fix is to make the check and the claim one synchronous step, by storing the work itself instead of a flag about it.

```
actor Syncer {
    private var inFlight: Task<Void, Error>?

    func sync() async throws {
        if let running = inFlight {
            return try await running.value
        }
        let task = Task { try await self.run() }
        inFlight = task
        defer { inFlight = nil }
        try await task.value
    }

    // run(): loadSettings(), then upload()
}
```

There is no `await` between reading `inFlight` and assigning it, so B cannot slip in between. B finds the handle and waits on the same work. We compiled both versions under Swift 6 language mode and ran two concurrent calls against each: the flag version performed two uploads that overlapped, and the task-handle version performed one.

Say the cost out loud, because the book's worked answer does: callers now share one outcome, including one failure.

> **THE ANSWER COVERS**
>
> The invariant, the line it breaks on, why the compiler stays silent, and what the fix makes callers share.

# Change one condition

A memorized answer is “replace the boolean with a task handle.” It will pass the first version of the question. The interviewer now changes the code, and a method behaves differently from a recitation.

**Move the settings read above the guard.** Suppose `loadSettings()` is awaited first, and then the guard and the assignment follow with nothing suspending between them. Is the original flag still broken?

It is not. With no suspension between the check and the set, those two lines run as one uninterrupted step, and a second caller will find the flag already `true` and return. The `await` on `upload` still admits other callers, but they are now stopped at the guard. The bug was never “booleans are bad” or “actors need task handles.” It was a suspension between the check and the act, and the skill being tested is locating it. This is why the strong version of the answer in the book opens with a question: does `sync()` await anything between the check and the set?

The new version has a different weakness. The second caller returns immediately, before the sync it was dropped in favor of has finished. If the caller needed the data to be fresh, that is a correctness bug of its own, and it is the reason the task handle is often the better design even when the flag is not broken.

**Now require independent retries.** Suppose each caller must be able to retry on its own failure, and one caller’s error should not become everyone’s error. The task handle shares one outcome by construction, so what was a clean fix becomes a policy conflict. The answer is no longer a pattern. It is a decision about what concurrent callers are allowed to share, and the book asks you to make that decision deliberately rather than inherit it.

Notice what stayed constant across both changes: the method. Mark each `await`, ask what can be true on the other side of it, and choose the smallest change that holds the invariant.

> **WHAT A SENIOR CAN TAKE FROM THIS**
>
> A fix is a pattern. A method survives a changed condition, because it starts from where the suspension is.

WHERE YOU STAND

## Ten minutes on your own code

The book's chapter 02 asks you to score yourself on observable behavior and attach evidence, because a rating without evidence is a feeling. This is the same idea, applied to one skill. Do it on paper, in order, with a timer.

1. **Find an actor you wrote.** If you have none, pick any `@MainActor` type or class with an `async` method that reads state, awaits, then writes state.
2. **Mark every `await`** in one method. Circle each one.
3. **For each circle, write the sentence:** "After this line, I can no longer assume ___." If you can fill the blank for every `await`, good. If you wrote "nothing," look again.
4. **Find one check-then-act** that spans a suspension. A flag, an optional cache, a dictionary lookup followed by a write. Write the two-caller timeline for it, line by line, as in the correction above.
5. **Say the fix aloud in under a minute,** including its cost. If you cannot name what concurrent callers now share, the fix is a recipe, not a decision.
6. **Answer without notes:** would Swift 6 strict mode have flagged the original bug, and why or why not?

### HOW TO READ THE RESULT

Now read your own results against the book's rows. If step 3 needed prompting from the code, that matches a 2 on dimension 1: you know the concepts and the implications arrive late. A 3 is a correct model with the normal failure path handled. A 4 is when the same pass also produced a test that gates the suspension and asserts one fetch, a cancellation story, and a sense of where else in the module the same bug will be.

Whichever number you land on, write down the evidence, not just the score. The session is only useful if next week you can see which `await` fooled you.

GET THE FULL BOOK

## You have just done what the book is for.

You took one question, found where a plausible answer breaks, corrected it, and watched a changed constraint test whether the correction was a method or a memory. That is one loop.

The full edition runs that loop across the whole surface a senior interview covers, and scores the result so you can tell preparation from practice. A diagnostic scored on six dimensions, 57 questions and 17 timeboxed drills, and twelve simulations, each with its prompt, a time plan, a scorecard reference, debrief questions and the drill to run next.

**The companion.** A Swift package of runnable drills and the harnesses behind the verified snippets, with an Xcode host app. `swift test` runs 23 tests.

**Get the full book**

**A reader's review of this book**

> ★★★★★
>
> I love this book. It's become my go-to starting point whenever I'm preparing for iOS interviews. Clear, practical, and packed with valuable insights.
>
> **Brian A., review on Gumroad**

Interview Runtime 1.2: iOS Interview Preparation, 380 pages, PDF and EPUB. Mike Salari, staff mobile engineer, 500+ technical interviews from the hiring side. Copyright 2026. Share this sample freely. Not affiliated with, authorized by or endorsed by Apple Inc.