FIRST EDITION

SwiftUI Under Load

Performance, concurrency, state, and migration for production Apple apps

MIKE SALARI

15+ YEARS IN MOBILE ENGINEERING · SALARI.DEV

READ THE FAILURE · FIX THE CAUSE · PROVE IT

A free sample of SwiftUI Under Load

What this book actually is

Your SwiftUI app works. The list scrolls, the previews render, the demo lands. Then it grows, and you meet the part of SwiftUI nobody taught you.

The list that was smooth at fifty rows stutters at ten thousand. Search returns yesterday's results because a slow request landed last. An image decode blocks the main thread at the exact moment a user is watching. Swift 6 strict concurrency turns the project into a wall of warnings you cannot read.

Most SwiftUI books stop at the happy path. This one begins where the happy path ends.

Every chapter opens on a failure you can reproduce, the exact symptom you would see in a running app. Then the mechanism underneath it. Then the smallest correction that works. Then the design that holds when the app doubles in size. Nothing here is a topic to memorize; it is a way of reading failure that makes you useful on a real team on day one.

What follows is one full chapter, unedited, so you can judge the book by the book.

Chapter 14: Lists with 10,000 items

The failure signal

Ledger's feed had 10,000 rows and used a `LazyVStack`, so the team believed the list was handled. Typing into search still froze the screen for hundreds of milliseconds per keystroke.

The container was lazy, which meant rows were built on demand. But every keystroke normalized, filtered, grouped, and sorted the entire dataset on the main actor before any lazy construction happened, and unstable ids then caused broad row replacement on top of that. The lazy container had solved the one problem it solves and left every other problem in the pipeline untouched.

"It is lazy" is not "it is fast."

The decision

Design the complete data-to-row pipeline, not just the container: stable identity, bounded snapshots, filtering done off the main actor and cached, pagination with a gate, cheap row bodies, and measured image work. A large list is a pipeline, and its performance is the performance of its slowest stage, which is almost never the container.

Stable identity is non-negotiable

Use a persistent domain id, never an index or a freshly generated id. Identity drives the diff, state preservation, animation, and task lifetime for every row. Unstable identity turns a one-row insert into a broad replacement, which is both a correctness bug and a performance one, because SwiftUI now believes many rows changed.

Filter outside the body

Normalize the query, debounce the intent, run the expensive filtering off the main actor when it is safe, and publish a small immutable snapshot. Cache by query and source revision, so re-typing a previous query is instant. The failure signal filtered all 10,000 records inside the update path on every keystroke. Moving that work off the main actor and behind a debounce is the fix. The container change was a distraction.

The minimal correction

Move the filter off the hot path and hand the list a bounded, stably-identified snapshot with a gated prefetch.

```
struct ActivityFeed: View {
    let snapshot: FeedSnapshot        // filtered, sorted, identified
        upstream
    let loadMore: @Sendable () async -> Void
    var body: some View {
        List(snapshot.items) { item in    // Identifiable by domain id
            ActivityRow(item)             // cheap body: reads finished
        values
                .task {
                    if snapshot.shouldPrefetch(after: item.id) {
                        await loadMore()  // gated upstream, one page in
        flight
                    }
                }
        }
    }
}
```

The store owns the derived snapshot, updated off the main actor when inputs change. The repository owns pagination behind a one-in-flight gate with id deduplication. The row reads finished strings and a prepared image, so its body is close to free. Each stage has a budget and a test, so “the feed is fast” becomes a set of verified claims, not a hope.

This was one failure. The book has thirty-six.

A few of the others you will recognize from your own code:

- A search box that returns results for a query the user already finished typing, because a slow request landed last. Eight lines fix it. Almost no app has those eight lines until someone puts them there.
- A home-screen widget that shows a different number than the app, both computed correctly, computed separately.
- A Swift 6 build that will not compile because a static property is not concurrency-safe, and the one-word fix that hides a real lesson.
- A Watch target added in a hurry that, six months later, disagrees with the phone about what a paused workout is.

Every one enters as a reproducible failure, and leaves as a fix you can prove.

Get the full version

salari.dev/books/swiftui-under-load/

Reader, the book. · Professional, the book plus the companion code, twelve labs and the staged app from broken to verified. · Complete, the book plus six production apps in full source you can rebrand and ship.

Mike Salari · salari.dev