FREE FIELD GUIDE

When the response never comes back

One lost response, two bookings, and the bigger problem a perfect fix still leaves behind

MIKE SALARI

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



Based on *Controlled Change: Architecture, Reliability, and Evolution for Mobile Systems*

AN ATLAS INCIDENT

The release was healthy. Support had two bookings for one trip.

A mobile release retries booking after a timeout. A small network cohort ends up with duplicate reservations. Crash-free metrics stay green. Support is the one that detects the pattern.

The retry should have been harmless, because booking carries an idempotency key. It was not, because the release generated the key in the transport adapter, so every retry left the phone as a brand new attempt. The first request had reached the server, or was still being processed when the phone gave up, and the phone never learned which.

The review afterwards found five gaps. None of them was a crash:

- the key was generated in the transport adapter rather than the durable command;
- there was no state for an outcome awaiting resolution;
- server ledger retention was too short for restored queues;
- the release guardrail measured request errors, not duplicate reservations;
- support tooling had no operation reference.

Only the first gap is about generating the key. Keep that in mind; it is where this guide ends up.

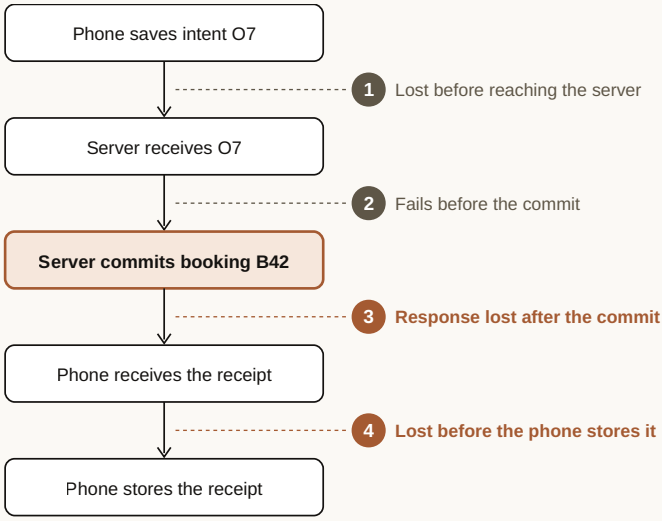
> The question to ask before you approve any change that writes to a server:
> **What happens when the response is lost after commit?**

Atlas is the fictional marketplace and field-operations product that runs through *Controlled Change*. It combines pressures common to mobile systems; its details are illustrative. Adapted from chapters 49 and 54.

WHY IT HAPPENS

A timeout tells you nothing about the outcome

A request can fail before reaching the server, after reaching it but before committing, after committing but before the response arrives, or after the response arrives but before local persistence.



Before a restart, 1, 2 and 3 look the same: no answer. After one, so does 4.
In 3 and 4 the booking already exists.

In cases 3 and 4 the server has already booked, and a timeout can also fire while the server is still working. A retry that looks new to the server books again.

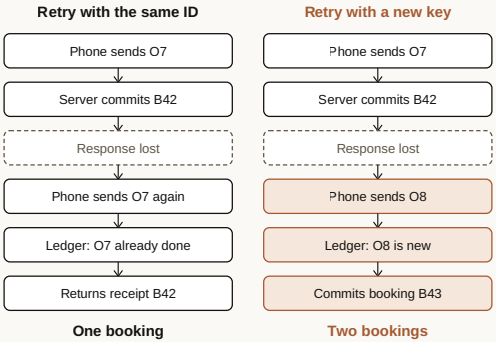
No queue or lease removes that uncertainty. A lease prevents concurrent processing, but exactly-once execution across a network boundary is usually an illusion. What works is at-least-once delivery, made safe by idempotent handling on the server.

Adapted from chapters 2, 20 and 21.

THE FIX

One attempt, one identity

An idempotency key identifies one logical attempt, not one tap or one network request. The client creates it before durable acceptance and reuses it for every replay; the server keeps a ledger of canonical results.



Born with the intent.
Before the intent is accepted, owned by the durable command. Not at the tap, and never in the transport adapter.

Claimed before the work.
A production ledger claims the key before it books, so only one request wins and a concurrent replay gets an in-progress status. The single-threaded simulator leaves this part to you.

Answered from the ledger.
A replay returns the stored outcome: the receipt, or a business rejection such as a conflict. Nothing runs twice.

Bound to its payload.
The companion simulator fingerprints account, operation kind, payload version and payload; a mismatch is rejected and the client quarantines the operation. A refreshed quote should be a new intent with a new ID.

Remembered long enough.
Longer than a phone can hold the operation, counting queue age and time offline. Past that window, look up the affected entity by its stable client reference instead of replaying.

Adapted from chapters 10, 21, 22, 49 and 54, and the companion sync simulator.

THE BIGGER PROBLEM

What a perfect key still cannot fix

An idempotency key answers one question: has this exact attempt already happened? Run the incident again with a perfect key and look at what is still open.

The release that minted new keys is still installed.

A client fix cannot reach phones that have not updated. Containment and old-version rules have to live on the server.

The screen already said Booked.

Completion truth belongs to the booking authority. The UI needs states for pending and unknown, or it reports a success the server never gave.

The process died with the request in flight.

The phone cannot tell whether the server saw it. Recovery needs one procedure that asks before it resends.

The dashboard could not see any of it.

The guardrail measured request errors. Duplicates are an invariant you measure from the authority, per release.

Two travelers change the same trip.

That is not a duplicate at all. It is concurrent valid intent, and deduplication does not solve it; a domain conflict policy does.

Support has no operation reference.

Repair means correlating payment and reservation records. Without an operation ID in the tooling, every case is archaeology.

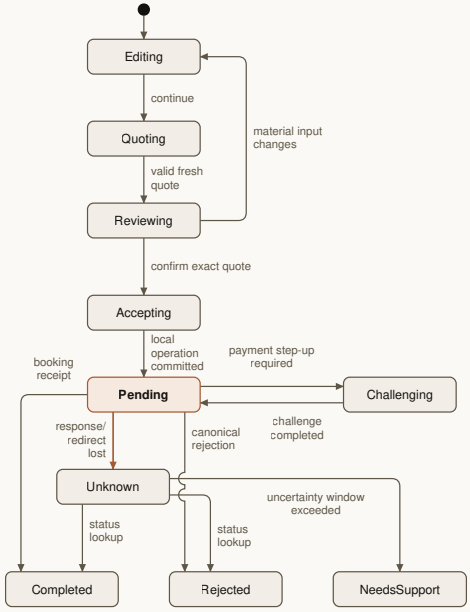
Each of these is a separate decision, and most have a different owner. The key made one failure safe. The system is no easier to change.

Adapted from chapters 16, 21, 22, 35, 49 and 54.

DECISION 1: THE SCREEN

Do not claim more than you know

In the Atlas checkout, the client owns the form and the durable intent, the booking service owns the receipt, and the UI owns presentation, not completion truth. So the client never marks checkout completed because the payment sheet closed successfully. It waits for the booking authority's canonical result, shows a pending receipt until then, and moves to Unknown when the response or the redirect is lost.



Checkout states, including step-up and an unknown outcome. Figure 54.2 in the book.

Booked screen after the payment sheet

Feels fast. Dishonest under delayed confirmation.

Pending receipt with a status lookup

Slightly more complex. Truthful and recoverable.

Adapted from chapter 54.

DECISION 2: RECOVERY

After a restart, ask before you resend

Anything essential after process death must be durable before the UI claims acceptance; a coroutine or a task is not. Two interruptions matter most:

The request had reached the server

Reconcile by status. Do not send it again.

The request never left the device

Send it again, with the same operation ID.

After a restart the phone cannot tell these apart. The companion sync simulator handles both with one procedure: work that was executing becomes Unknown, the engine asks the server for the operation's status, and it resends with the same ID only when the server has no record of it.

The resend is safe because of the ID, not the lookup. A no-record answer can be wrong, because the original may still be in flight. With the same ID, a ledger that claims each key before it books turns a late arrival into a replay, not a second booking. Trust the answer only when the lookup reads the authoritative ledger, not a cache or replica, and only inside the ledger's retention window; past it, look up the affected entity by its stable client reference.

If that still leaves doubt, pull authoritative changes, replay only when idempotency is guaranteed, and escalate to a visible support state. Never silently create a second payment, booking, or message thread to make the UI look resolved.

To test it, end the process. Recreating a view model in the same process proves nothing. Drive the operation to a named transition, use `adb shell am kill` on a backgrounded Android app or `terminate()` then `launch()` in an iOS UI test, relaunch cold, and check the screen and the server's count of effects.

Adapted from chapters 16 and 21, and the companion sync simulator.

PROOF, NOT REASSURANCE

The first case, as a test you can run

This test is copied byte for byte from the sync simulator in the Extended edition. The request reaches the simulated server, the process dies, and a new engine starts from the operation's durable state.

```
    def test_death_after_server_commit_reconciles_without_resending(self) -> None:
        server = SimulatedServer()
        op = operation("21")
        op.state = OperationState.EXECUTING
        server.submit(op)                       # the request reached the server, then
the process died

        engine = SyncEngine.after_process_death(server, [op])
        self.assertEqual(op.state, OperationState.UNKNOWN)
        engine.run_until_idle()
        self.assertEqual(op.state, OperationState.ACKNOWLEDGED)
        self.assertEqual(server.side_effect_count[op.operation_id], 1)
        self.assertEqual(engine.execution_order, [])   # reconciled by status, never
resent
```

From `sync-simulator/tests/test_sync_engine.py`, one of its eight tests. Long lines wrap to fit the page.

The server shows one side effect, which its ledger would guarantee even after a resend. The decisive assertion is the last one: the execution order is empty, because the new engine asked by identity and sent nothing.

The other seven tests cover the neighbors of this failure: a response lost after commit without a restart, duplicate delivery, a drop before the server, the second process-death case, a stale revision that becomes an explicit conflict, dependency ordering, and a poison operation that must not block unrelated work.

BEFORE A BOOKING CHANGE SHIPS, ASK FOR EVIDENCE OF

1. one domain effect when one operation is replayed;
2. a rejection when the same key arrives with a different payload;
3. a timeout after server commit ending in one booking;
4. a restart test at every transition of the operation;
5. old-client fixtures for new statuses;
6. a kill-switch drill during pending and new journeys.

Adapted from chapters 16, 21 and 54. Test from the Extended edition's companion code.

DECISION 3: THE SERVER

The fix ships. The bug stays installed.

A backend can deploy in minutes. A mobile client may remain active for months, sit offline for weeks, or queue an operation before upgrading.

So the corrected release closes nothing on its own. The build that minted a new key on every retry is still on phones, and some of those phones hold operations they have not sent yet. Atlas contained the incident at the server: it stopped new booking commands from the affected client version and showed a status lookup. Repair correlated payment and reservation records, cancelled duplicates, and contacted affected users.

The book's version-skew chapter names the week teams forget: a traveler opens an old client after a month offline, and a queued old-format request is sent. Fresh traffic on the old version may have stopped weeks earlier, while operations created before the upgrade are still waiting on devices.

What the server owns for the long run. It supports old request versions through a published support window and can deny unsafe operations with a stable reason. Old clients decode unknown values safely: a safety-critical unknown state blocks duplicate action and triggers a status refresh or upgrade guidance.

Where the fix stops. A ledger stops replays of one attempt. It cannot help once the key has expired or the state has moved on, and set quantity to 3 is easier to replay than increment quantity. And checkout is not fully offline: for scarce inventory, save the details, notify when the connection returns, then get a fresh quote and confirmation.

Adapted from chapters 21, 22, 49 and 54.

The next failure will not look like this one

Atlas meets the same shape of failure elsewhere in the book. Every time, the dashboards look fine and the problem surfaces through support or the server's own records.

> **Release 9.2** CHAPTER 35
>
> The dashboard was the best it had been all quarter, because a refactor had moved the telemetry call and operations stuck in Unknown reported nothing. On the server ledger, operations reaching a terminal state within an hour fell from 99.6 to 97.1 percent.

> **The general conflict engine** CHAPTER 52
>
> An offline booking change was merged locally and shown as confirmed. The server rejected it against inventory, and support found travelers holding screenshots of changes that never existed.

> **The remote layer that became a language** CHAPTER 52
>
> A new server-driven action reached clients two versions old, and they silently skipped a required onboarding step.

You will not predict the next failure. You can decide whether fixing it will be cheap. Atlas reversed three decisions made for good reasons, and the book explains why it could:

> *Each one was reversible because Atlas had already paid for the disciplines this book argues for: [...] operations with identity so booking could move to server authority without data loss [...]. Good architecture did not predict the future. It made being wrong affordable.*

The identity that stops a duplicate booking is the same one that later lets booking move to server authority. That is the book's question in one example: how cheaply and safely can this product absorb the next important change?

Numbers are illustrative. Adapted from chapters 1, 35 and 52; the quotation is from chapter 52.

INSIDE THE BOOK

Six situations you will face, and what you leave with

The platform team wants every capability split into api, impl, ui and data modules.

Chapter 52 shows the change-coupling evidence that made Atlas merge them back, and the rule that survived: boundaries follow measured change coupling and real owners, not a template.

Product changes cancellation, and two old releases are still in the field.

Chapter 22 plans it week by week, from the first server change to a typed rejection with an upgrade path after the support window, including the queued request that arrives after a month offline.

Someone proposes one engine for every sync conflict.

Chapter 21 separates duplicates, concurrent intent and version incompatibility, and chapter 52 shows what happened when booking went through a generic merge.

Leadership asks to let coding agents merge dependency bumps without review.

Chapter 60 qualifies one task class on held-out cases: 96 auto-merges, all good, still leave the reliability bound at about 96.9 percent against a 97 percent target, and the team says so instead of moving the target.

The framework, the date and the vendors were fixed before anyone asked you.

Chapter 61 classifies every consequence by reversibility and decides everything the imposed decision left open.

You have to approve a design by Friday.

Chapter 1 gives you six changes to walk through it first, from inventory becoming reservable for ten minutes to a payment SDK removed within one quarter. Nine appendices hold the canvas, 12 example ADRs, the review rubric and 10 migration playbooks.

Numbers are illustrative, as in the book.

CONTROLLED CHANGE

You fixed one failure. Now make the next change cheap.

Book

Current price on the store

MAKE THE DECISIONS.

The complete book in PDF and EPUB: 505 pages, 61 chapters, nine appendices for architecture reviews.

Pro

Current price on the store

SEE THEM IN RUNNING CODE.

Book, plus the Atlas reference system: five projects with tests (iOS, Android, Kotlin Multiplatform, backend, contracts), an 83-page toolkit and a 19-page workbook.

Extended

Current price on the store

BREAK IT ON PURPOSE.

Pro, plus React Native, Flutter and the sync simulator from this guide: its eight tests, ready to run and break.

Buy on Gumroad **Visit salari.dev**

All three editions are on one page; you pick the edition before you pay.

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