The Signups That Never Arrived

Reliability hardening for a 21-function, event-driven Azure platform — without the rewrite nobody needed.

At a glance

The challenge. A choreographed, event-driven backend — 21 Azure Functions reacting to Service Bus messages — had grown one handler at a time. Three different ways of acknowledging a message meant delivery guarantees varied function to function, and the profile-creation path could silently drop a new registration.

The work. A reverse-engineered architecture baseline, one standardized message-settlement pattern rolled across every function, an at-least-once redesign of the create path, poison-message dead-lettering by default, and consistency observability for the CQRS read models.

The outcome. Dropped registrations closed at the source, uniform settlement with proper dead-lettering across all 21 functions, a bounded and observable read-model divergence window, and a versioned architecture baseline the team actually owns — with no new Azure spend.

A registration that was never there

Somewhere in the logs there's a message that got acknowledged and then went nowhere.

A new member signed up. The platform said "got it" to the message broker — and then the call that was supposed to actually create the profile failed. Because the message had already been marked complete, nobody retried it. Nobody dead-lettered it. Nobody knew. The signup didn't throw an error. It didn't fire an alert. It simply never happened, and the only evidence it ever existed is a gap you'd have to already suspect to go looking for.

That's the kind of bug that never shows up in a demo. It compiles. It passes tests. It works every single time you watch it. It only fails when you aren't looking — which, for a registration path, is most of the time.

The client runs a high-traffic, multi-site community platform: seven sites on one codebase, with a backend built the modern way. Web apps publish messages onto Azure Service Bus, and a fleet of independently deployable Azure Functions react to them asynchronously. Twenty-one of them. It's a good architecture — loosely coupled, independently scalable, exactly the shape you'd draw on a whiteboard. The problem was never the shape. The problem was that it had

grown one function at a time, and no two of them agreed on what to do when something went wrong.

The real problem isn't a bug. It's a missing decision.

When you acknowledge a message is not a coding detail. It is an architectural decision about delivery guarantees, and it has to be made once, on purpose, for the whole system — not twenty-one times, by accident, one function at a time.

By the time I was engaged, three different patterns had quietly emerged across the fleet:

- **Complete first, then work.** Fast, and at-most-once: if the work throws after the acknowledgment, the message is already gone. Perfectly fine for a page-view counter. Not fine for a registration.
- **Work first, then complete.** The message is acknowledged only after the work succeeds, and the runtime retries on failure. At-least-once — the right default for anything that matters.
- **Full settlement, done properly.** Exactly *one* function did it end to end: complete on success, abandon on a transient error so it retries with backoff, and dead-letter a poison message so it stops looping and starts being visible.

The gap between those three is the entire story. The profile-creation path was in the first bucket. That is the silently-dropped signup, stated in one sentence.

Discovery: map before you modernize

There was no current architecture document. Nobody could point to the one picture that showed how the pieces fit — which meant every conversation about "the reliability problem" started with somebody reconstructing the system from memory.

So the first deliverable wasn't code. It was a map. I reverse-engineered the whole landscape from source: every function, every Service Bus topic, every store, and the read-model fan-out that ties them together.

What the map showed:

- **21 functions** across four jobs — audit and analytics, notifications, read-model projection, and lifecycle housekeeping.
- **Service Bus topics as the spine.** Producers and consumers share only a message contract; either side can scale or fail on its own.
- **Polyglot persistence.** SQL for audit and config, Cosmos DB for profiles and read models, Redis for cache, Blob for templates.

- **A CQRS fan-out.** One write to the canonical profile document projects into five denormalized, query-optimized read models — username lookup, geo/filter search, birthday lists, newest members, tag search — each partitioned for its own query.
- **External integrations.** Transactional email and SMS, an external profile API, and telemetry.

That baseline — a C4 context view, an entity map of the Cosmos containers, and sequence diagrams for the message lifecycle — became the reference everyone had been missing. It's the thing you point the next developer at, instead of making them read twenty-one function apps to understand one.

What discovery surfaced

Four findings, in the order they mattered:

- **CRITICAL — Registration-loss risk.** The create path acknowledges the message before the downstream call succeeds. A failure after that point drops a signup, silently, with no retry and no alert.
- **CRITICAL — Inconsistent settlement.** Three completion strategies across 21 functions means delivery guarantees you can't reason about, because they're different everywhere you look.
- **IMPORTANT — Read-model divergence.** The five projections run independently. A partial failure leaves search and member lists temporarily disagreeing with the source of truth — with no visibility into when, or for how long.
- **IMPORTANT — No standard poison handling.** One function dead-letters correctly. The rest either abandon-loop or drop. A poison message is therefore either an infinite retry or a silent loss — never an alert.

The decision: harden it, don't rewrite it

This is the point in a lot of engagements where a certain kind of consultant sells you a rewrite. Event sourcing. A new orchestration layer. A service mesh to "manage" the functions. A quarter of net-new platform to solve a problem the platform didn't have.

The architecture was already right. Event-driven was the correct choice for this workload. Service Bus was the correct backbone. The read-model projections were a textbook CQRS fan-out. None of that needed to change — and reaching for a re-platform would have been resume-driven development on the client's dime. What the system needed wasn't a new architecture. It needed the one it already had, applied consistently.

So the scope was deliberately narrow: **standardize the thing that was inconsistent.** Nothing more.

The work

- **One settlement pattern, everywhere.** Complete on success. Abandon transient errors so the runtime retries with backoff. Dead-letter poison messages so they stop looping and start being visible. The pattern already existed in the codebase — one function had it right. The work was making it the standard and rolling it across the other twenty.
- **The create path, made safe.** Redesigned to at-least-once, end to end: the message isn't acknowledged until the profile actually exists. A transient failure now retries; a permanent one dead-letters where someone can see it. The silently-dropped signup stops being possible.
- **Poison handling, by default.** A uniform dead-letter-plus-retry policy, so every function fails the same visible way instead of each inventing its own.
- **Read-model consistency, made observable.** Instrumentation on the five projections so divergence is bounded and, more importantly, *visible* — you can see when a read model is behind, instead of hearing about it from a confused member.
- **The baseline, committed.** The architecture document, the diagrams, and the settlement standard — committed to the repo, versioned, and reviewed like code. The reason the system looks the way it does now lives in a file with a commit history, not in anyone's head.

What the client got

- **A registration path that can't silently drop a signup** — the failure mode that started the conversation, closed at the source.
- **Uniform message settlement across all 21 functions,** with poison messages dead-lettered instead of looping forever or vanishing.
- **Read-model divergence that's bounded and observable,** not a mystery that surfaces through a confused user.
- **A versioned architecture baseline** — C4 diagrams, an entity map, and sequence flows — that the team owns and extends, instead of a system only its original authors understood.

And no new Azure spend. The work hardened what already ran; it didn't add infrastructure to pay for and operate.

The practical version

If your event-driven system grew one handler at a time, some of this will sound familiar. What I'd take from it:

- **Settlement is an architecture decision, not a per-function detail.** Decide your delivery guarantee once, write it down, and make every handler conform.
- **"It works" and "it's safe" are different claims.** The dangerous failures are the ones that complete successfully and lose the work anyway. For every handler, ask: if this throws *after* the acknowledgment, who finds out?
- **Dead-letter — don't abandon-loop, and don't drop.** A poison message should become an alert, not an infinite retry or a silent gap.
- **Map before you modernize.** If there's no current architecture document, that's the first deliverable — and often it's the one that reframes the whole problem.
- **The best modernization is usually the smallest one.** Right-sizing applies to fixes too. The correct pattern applied consistently beats a new architecture almost every time.