

Redundancy Without Reliability

A consumer VPN needed automatic failover if one server went down. Instead of building it, I tested whether the simple version already worked — and the result overturned the plan, placing the real constraint in a layer I did not control. What follows is how I reasoned, tested, and decided, and what I ultimately chose not to build.

THE PROBLEM I SET OUT TO SOLVE

I run a small consumer VPN: a customer installs an app, taps connect, and their traffic is routed through one of several servers I operate in different regions. I wanted to add automatic failover — if the server a customer was on went down, they would move to a healthy one without noticing. It is the kind of reliability feature that separates a service people trust from one they quietly abandon after the first bad evening.

I should be straightforward about my starting point: I have no formal background in networking or computer science. What I had instead was a method — work through unfamiliar territory one decision at a time, refuse to add complexity without a reason, and verify each step before building the next. Throughout, I used an AI assistant (Claude) as a reasoning partner: somewhere to pressure-test my thinking and have unfamiliar mechanics explained, while keeping the testing, the decisions, and the final calls with me.

HOW I WORKED THROUGH IT

The first useful move was not technical. It was catching an assumption I had never actually checked.

I had believed my central control server — the machine that manages users and coordinates the rest — could reroute customers when a server failed. Working it through, that was wrong. The control server never carries customer traffic; it only issues instructions. The real connection runs directly from the customer's app to a specific server, so if that server dies, there is nothing in the middle to redirect them. In plain terms: the dispatcher knows a taxi has broken down, but the passenger is already inside it, and the dispatcher cannot teleport them into another cab.

That reframing mattered, because it meant failover could not be solved where I had assumed it would be. Before designing anything, I wanted to know what actually happened today — with no new machinery — the moment a server died.

TESTING THE ASSUMPTION INSTEAD OF TRUSTING IT

The common belief here, widely repeated and rarely checked, is that simply listing two servers for a customer gives you automatic failover: if one is unreachable, the app moves them to the other. My entire plan rested on it. The advice I leaned on, and agreed with, was to test that belief directly before building anything on top of it — and to do it on a spare server with no real customers, so a failure cost nothing.

So I ran a controlled failure. I deliberately disabled one server at the network level, reproducing a genuine outage while holding every other variable constant, then repeated the identical failure across four different VPN client apps a customer might realistically use, isolating the app as the single variable. I judged the result only by whether a live connection actually recovered — not by the apps' own status indicators, several of which were unreliable in this configuration and marked healthy servers as unreachable. That detail alone justified the exercise: the tools were lying, and only an outcome-based test caught it.

WHAT THE TEST REVEALED

Three results held across every trial.

- **No automatic recovery.** In none of the four apps did the connection move to the healthy server on its own. The customer was simply stranded on the dead one.
- **The deciding layer was outside my control.** Which server a customer connects to is decided by the app on their own phone — software I cannot modify. However well I engineered the servers, I could not guarantee the outcome, because the component making the decision was never mine to command.
- **Where switching did happen, it was blind.** One app offered a manual switch but picked the next server by list order, ignoring location and health — an unpredictable jump rather than a managed one.

The lesson underneath all three: I could build a solution, test it in isolation, and prove it worked — then still watch it fail for real, because the layer that decides was never mine to control. “Works when I test it” and “works for a customer, across every layer I don't own” are simply different claims.

WHY I DECIDED NOT TO BUILD IT

This is the decision I am most confident was correct, and it cuts against the instinct to build.

Because the deciding layer lived on the customer's device, no amount of server-side engineering could guarantee the behaviour end to end. The elaborate failover system I had planned would most likely have failed silently in production — the worst kind of failure, since it looks fine right up until a customer needs it. And the honest business

reality was that the service did not need it: a customer whose server dies already sees a second entry in the app and taps it. The gap I was about to spend weeks closing was one almost no one would ever notice.

So I stopped. I kept the redundant pair as a manual fallback and redirected the effort toward things genuinely within my control. The whole conclusion cost a single afternoon of testing on infrastructure with no live customers — instead of the weeks that would have gone into building something engineered to fail.

WORKING WITH AN AI, AND WHAT STAYED MINE

Since the collaboration was part of how I worked, it is worth being precise about the division of labour. I used the assistant to move quickly through territory I did not know — explaining unfamiliar mechanics, sanity-checking reasoning, and suggesting what to test and in what order. When its suggestions did not match what I was actually seeing, I pushed back and trusted the real result over the confident answer.

What did not come from the tool was the part that mattered. Choosing which question was worth asking, running the test on real infrastructure, reading the outcome honestly, and making the call to stop — those stayed with me. Producing a configuration is cheap now; any tool can generate one. Deciding which problem is worth solving, and recognising when the right answer is to build nothing, is the part that cannot be automated. That is the part I kept.

WHAT GENERALISES

Three things travel well beyond a VPN. Verify a load-bearing assumption with the cheapest possible test before building on it — an afternoon of breaking something on purpose can save weeks of building the wrong thing. Map which parts of a system you genuinely control, because a component that works in isolation is not a system that works end to end. And use new tools to accelerate your reasoning without surrendering your judgment — let them help you think faster, but keep the decisions, especially the decision to stop, for yourself.