Beyond Localhost · Follow publication

Member-only story

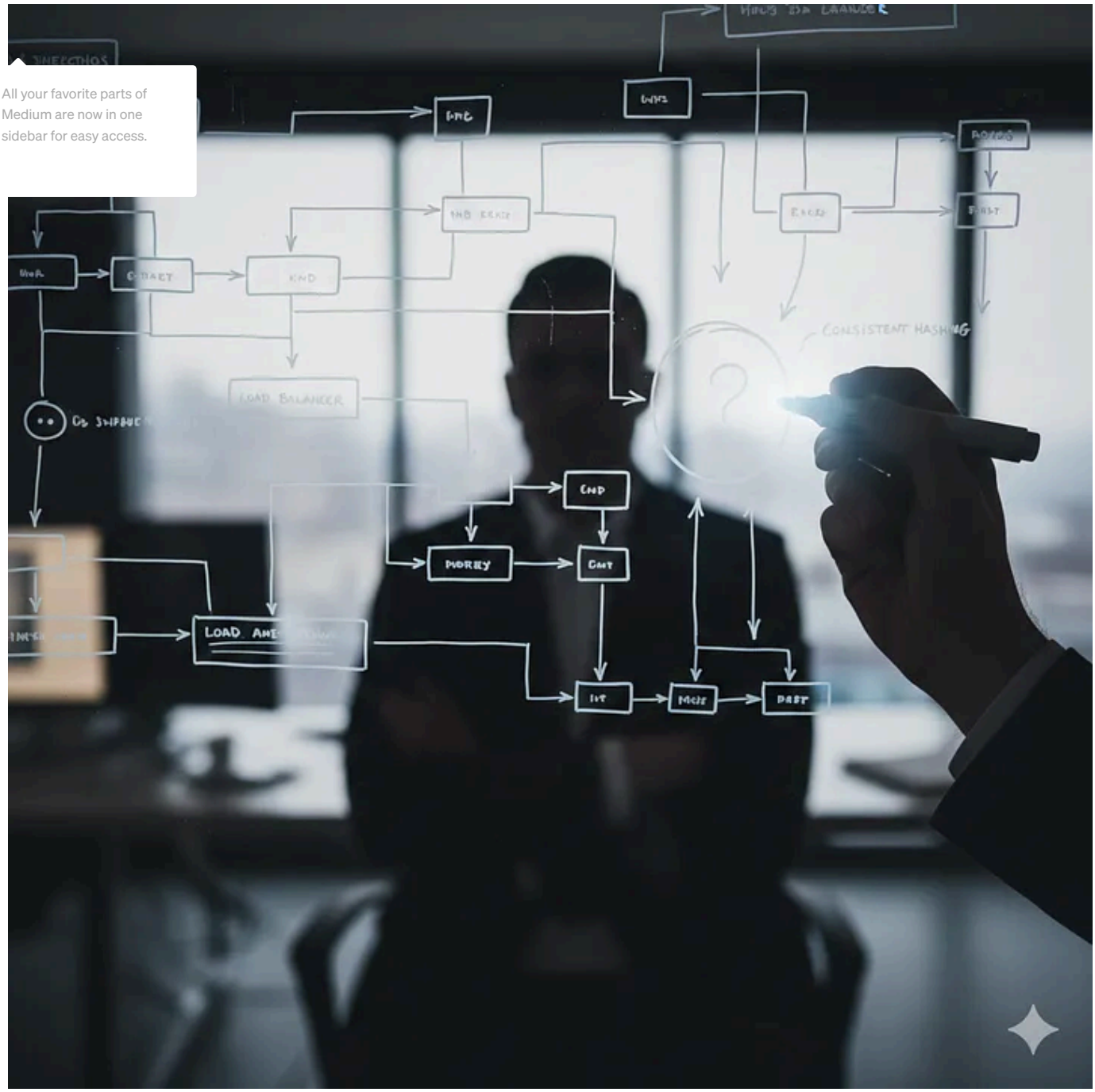
I Thought I Knew System Design Until I Met a Google L7 Interviewer

A single whiteboard question revealed the gap between knowing patterns and actually designing systems that scale.

6 min read · Dec 22, 2025

The Speedcraft Lab Follow

Listen Share More



Patterns get you started, Numbers decide what survives

The marker was in my hand. The problem seemed straightforward enough: design a URL shortener. I'd done this before, or at least I thought I had. Hash the URL, store it in a database, return the short code. Classic stuff. I started drawing boxes for the API layer, the cache, the database. The interviewer just watched, nodded a bit, then asked: "What happens when you hit 10 million writes per second?"

I completely froze.

Not because I didn't know about sharding or replication. I'd spent weeks studying those patterns. I froze because in that moment I realized something kind of uncomfortable: I'd been memorizing architectures without actually understanding why they existed in the first place. Like learning to recite a poem in a language you don't speak.

You know that feeling when you suddenly see yourself from the outside? That was it.

Turns out I'm not alone in this. There was this survey last year of about 800 engineering candidates, and something like 67% could draw a load balancer perfectly, but only 19% could actually explain when horizontal scaling stops being the answer. Which is wild when you think about it. We're all walking around with these mental diagrams, but the *why* behind them is fuzzy.

By the end of this you'll understand why system design interviews feel like you're suddenly speaking a different language, and what actually changes when you stop memorizing patterns and start reasoning about trade offs in real time.

ı carrying this mental library of reference architectures. Instagram's feed system. Netflix's CDN strategy. . I genuinely thought mastery meant collecting more of these patterns, like Pokemon cards or something.

The interviewer smiled when I mentioned consistent hashing. Seemed pleased even. Then he asked: "Why consistent hashing here though? What problem does it solve that a simple modulo wouldn't?"

I started to answer. Then stopped. Wait. Why *was* I using consistent hashing? Honestly? Because I'd seen it in every single "design a distributed cache" tutorial I'd ever read. It was the pattern everyone used. But for a URL shortener, with deterministic keys and fairly predictable traffic patterns, did I actually need the complexity of ring based partitioning?

Or was I just pattern matching without thinking?

That's the thing nobody tells you. The gap isn't about knowledge. It's about this reflex we develop to reach for patterns before we've even understood what we're actually trying to solve. Senior engineers, the ones who've been doing this for years, they don't start with architecture diagrams. They start with numbers. Boring, unglamorous numbers. How many users? What's the read to write ratio? What's going to break first when this thing gets real traffic?

The boxes and arrows come later, after the math basically forces your hand.

When Everything Shifts

Here's where it got interesting. The interviewer stopped asking "how would you design this" and started asking "what happens when."

What happens when your primary database goes down right in the middle of a transaction? What happens when cache invalidation lags by 30 seconds during some viral spike? What happens when two datacenters split and both of them think they're the primary?

These aren't gotcha questions, which is what I thought at first. They're literally how production systems fail in real life. There was this analysis done last year of something like 200 outages at major tech companies, and about 73% involved some kind of state inconsistency during partial failures. The exact scenarios most of us never even think to rehearse.

The move that helps: pick literally any component in your design and force yourself to think through three ways it could fail. Database? What if writes succeed but reads are lagging behind? Cache? What if things get evicted faster than they can be repopulated? Load balancer? What if health checks are passing but the service is actually deadlocked inside?

Walk through each scenario. Out loud if you can. If you can't articulate what breaks and how you'd even detect it, you're probably not ready yet.

The Thing About Trade Offs

I used to think the goal was proposing a "correct" architecture. Redis for caching, PostgreSQL for strong consistency, Kafka for event streaming. Safe choices. Defensible. By the book.

The interviewer leaned back in his chair. "Okay. Now design the same system, but you can't use a distributed cache."

I just blinked at him. Every system design I'd studied *depended* on caching to hit any kind of scale. Without it, wouldn't the database just collapse under load?

He waited. Patient. Then: "What if your cache hit rate is only 40% because URLs follow a long tail distribution and barely ever repeat? Does caching still help at that point, or does it just add latency and operational headaches?"

Oh.

This is the part that changes everything. Every component has this envelope of conditions where it actually helps. Caching works when reads dominate and access patterns cluster together. Sharding makes sense when write throughput matters way more than transactional guarantees. Replication helps when read availability beats consistency.

But none of these are *always* true. The best answer completely depends on the actual numbers you're working with, and whether you can defend the math.

ıte your assumption out loud. "I'm assuming like a 95% cache hit rate because URLs probably follow a . If that assumption is wrong, this whole design breaks." Interviewers actually reward that kind of honesty nfidence.

Coming Back Around

The marker was still in my hand. The whiteboard was completely full by now, but I wasn't done. I erased one box, the distributed cache, and redrew the whole flow. Direct database reads for the first million requests. Add caching only after you've proven the hit rate actually justifies the complexity. Start with vertical scaling until the math literally *forces* you toward horizontal.

The interviewer nodded. "Now you're designing."

It wasn't about knowing more patterns. It was about defaulting to the simplest possible thing that could work, then letting real, measured constraints push you toward complexity. Three things that seem to hold:

Start with one database, one server, no cache. Only scale when some specific metric like latency or throughput or cost crosses a threshold you can actually name. When you add a component, explain what new failure mode it introduces and how you'd even detect it.

Every box you draw should solve a problem you've already proven exists.

This is what senior engineers do in actual production. They don't over design. They instrument things, measure, and evolve based on what they learn. Same discipline applies in interviews.

What Actually Changes

System design interviews aren't really about drawing the "right" architecture. They're about showing you can reason when things are uncertain, defend trade offs with actual numbers, and adapt when constraints shift.

What matters is training yourself to question every default choice. Why this database? Why this cache? What breaks first? The answers don't come from memorizing more diagrams. They come from working backward from failure modes and forward from constraints until the design feels forced, not chosen.

Small move that helps: take any system design you've practiced. Pick one component. Just remove it. Can the system still work? If yes, you didn't actually need it. If no, what metric proves it's necessary?

That's how you build judgment instead of just pattern fluency.

What's one design decision you've been making by default, without really defending the trade off?

Follow me for more such content.

System Design Interview | Software Engineering | System Design Concepts | Scalability | Architecture