Cloud & Architecture

Technical Guide

Prepared by

Kadharmoideen Fadurudeen

In this guide

2 articles · ~28 min total reading

Cloud & Architecture | April 20, 2026 | 13 min read

Serverless vs. Traditional Backends: A Practical Decision Framework

Cold starts, cost curves, and operational overhead — a practical framework for deciding when serverless actually wins over a traditional always-on backend, and when it quietly costs you more.

Few architecture debates get re-litigated as often as "serverless vs. traditional backends" — and few get answered as badly. Most takes are context-free: someone had a great experience with Lambda on a spiky internal tool and now recommends serverless for everything, or someone got burned by cold starts on a latency-sensitive API and now avoids it entirely. Both are right about their own system and wrong as general advice.

After 19+ years building and operating backends — some serverless, some on containers, some on bare metal I had to rack myself early in my career — I've stopped thinking about this as a technology choice and started thinking about it as a traffic-shape and team-shape choice. This is the framework I actually use.

The one-sentence version

Serverless is a bet that your traffic is spikier than your team's ops capacity can comfortably provision for. If that bet is true, serverless wins. If your traffic is steady and predictable, a traditional backend almost always wins on cost and latency — you're just paying for compute you're not using in a different, less visible way.

Where Serverless Genuinely Wins

I reach for serverless (Lambda, Vercel Functions, Cloudflare Workers, Cloud Run with scale-to-zero) by default in a few very specific situations, and it's rarely close:

- **Spiky, unpredictable traffic.** Webhook receivers, form submissions, scheduled jobs, image/video processing triggered by uploads — anything that goes from zero requests to a burst and back to zero. Provisioning a server to sit idle for this is pure waste.
- **Small teams with no dedicated ops capacity.** No patching, no capacity planning, no 3am paging for a disk filling up. For a two-person team shipping an MVP, that operational silence is worth real money even if the per-request cost is technically higher.
- **True pay-per-use at low-to-moderate sustained volume.** If you're serving a few hundred thousand requests a month, not tens of millions, the free tier and per-invocation pricing of most serverless platforms will beat the cost of an always-on instance, full stop.
- **Independent scaling per endpoint.** One route gets hit 100x harder than the rest during a launch. Serverless scales that one function without you having to think about it; a monolith on a fixed fleet scales everything together or nothing at all.

Where It Quietly Costs You More

The costs that don't show up in the marketing page are the ones that matter most, because they're the ones teams discover after they've already committed:

Cold starts

A function that hasn't run recently pays a startup tax — sometimes tens of milliseconds, sometimes multiple seconds if it's pulling in a large dependency tree or running on a language runtime with a heavier boot time. For a background job, nobody notices. For a user-facing API on the critical render path, that's a visible stutter, and it happens at the worst possible time: right when traffic is picking up and instances are scaling out.

Execution time limits

Most serverless platforms cap execution duration. That's fine for a typical API call, but it forces awkward workarounds for anything genuinely long-running — large report generation, video transcoding, bulk data migrations. Teams end up building a secondary queue-and-worker system just to route around the limit, which means they're now operating two backend paradigms instead of one.

The cost curve flips at scale

Per-invocation pricing is a great deal when you're not using much compute. It stops being a great deal once your traffic is high and steady, because you're paying a premium, per request, for elasticity you're no longer using — the load isn't spiky anymore, it's a flat line. Past a certain sustained volume, a couple of reserved, right-sized instances running the same workload 24/7 is reliably cheaper.

Lock-in on proprietary triggers

Once your business logic is wired directly into a provider's event model — queue triggers, storage triggers, proprietary orchestration — moving to another platform isn't a redeploy, it's a rewrite of your integration layer. A containerized service behind an HTTP interface is portable almost by default; a function tightly coupled to one cloud's event bus usually isn't.

A Practical Decision Framework

When I'm asked to make this call on a new project, I score four dimensions. None of them alone is decisive — it's the combination that points to an answer.

Dimension	Favors Serverless	Favors Traditional Backend
Traffic pattern	Spiky, bursty, unpredictable, or near-zero idle periods	Steady, predictable, sustained near-constant load
Team / ops capacity	Small team, no dedicated infra/ops role	Team with infra experience or a platform team
Latency sensitivity	Tolerant of occasional cold-start latency	Hard real-time or user-facing critical path
Sustained request volume	Low to moderate	High and consistent enough to saturate reserved capacity

If three or more rows point the same direction, that's your answer. When it's a genuine split — spiky traffic but latency-critical, for example — that's usually a signal you need the hybrid approach below rather than a single paradigm for the whole system.

The Hybrid Approach Most Real Systems End Up With

In practice, the systems I've built rarely pick one paradigm exclusively. The pattern that shows up again and again:

Traditional core, serverless edges

An always-on service (containers on ECS/GKE/a plain VM fleet) handles the steady-state hot path — the main API, the database-heavy reads, anything latency-sensitive. Serverless functions sit around the edges for exactly the workloads they're built for: webhook ingestion, scheduled cron jobs, thumbnail generation on upload, PDF exports, one-off data backfills. Each piece runs on the paradigm that fits its actual traffic shape instead of forcing one model onto the whole system.

This isn't a compromise — it's usually the more disciplined design. It forces you to actually think about which parts of your system are spiky and which are steady, instead of defaulting to whatever's trendy for everything.

What I Tell Teams Asking Me to Decide

- Measure your actual traffic shape before picking a paradigm — don't guess, pull the numbers.
- If you don't know your sustained volume yet (pre-launch), start serverless — the downside of switching later is smaller than the downside of over-provisioning for traffic that never shows up.
- If you're migrating an existing system with known, steady load, model the cost at your real request volume before assuming serverless is cheaper — run the numbers, don't trust the pricing calculator's default assumptions.
- Keep your business logic decoupled from provider-specific trigger code so you can move the boundary later without a rewrite.

The Honest Closing

There is no universal best practice here, and anyone who tells you otherwise is generalizing from one system. Serverless and traditional backends aren't competing philosophies — they're tools tuned for different traffic shapes and different team constraints. The job isn't to pick a side; it's to be honest about your actual traffic pattern, your team's ops capacity, and your latency requirements, and let those numbers make the decision for you.

Cloud & Architecture March 12, 2026 15 min read

Designing for Scale: Lessons from Serving Millions of Requests

What actually breaks first when traffic grows 100x — caching layers, database contention, and the architecture decisions that are cheap to make early and expensive to retrofit later.

Over 19+ years, I've been on call for systems the first time they crossed a few hundred requests a second, and I've been on call for systems the first time they crossed a few hundred thousand. The failure pattern is almost never a surprise once you've seen it a few times — traffic grows, and things break in a fairly predictable order. What surprises people isn't which layer fails, it's how much time and money gets spent hardening the wrong one first.

This isn't a theory piece about scalability patterns. It's the order in which I've actually watched systems fall over, and what I now build in from day one because retrofitting it later is so much more expensive than it looks on a whiteboard.

The pattern I keep seeing

Teams over-invest in the layer that's easy to reason about (usually "we need microservices" or "we need a fancier database") and under-invest in the boring stuff that actually falls over first: connection pools, N+1 queries, and a single server quietly holding session state that nothing else can see.

What Breaks First: The Database

Almost every scaling incident I've been paged for traces back to the database before anything else. Not because the database is poorly chosen — because it's the one component every request eventually has to go through, and it's the one that's hardest to horizontally scale without real architectural work.

Connection Pool Exhaustion

At low traffic, a connection pool of 20-50 connections is invisible — requests come and go fast enough that nothing ever queues. The moment traffic multiplies, or a single slow query starts holding connections a few hundred milliseconds longer than usual, the pool empties out and every subsequent request queues behind it. From the outside this looks like your entire application going down. From the inside, it's one query that got 300ms slower.

> **⚠ A real one**
>
> I once watched a checkout flow start timing out under load, and the root cause was a single reporting query someone had added to a request handler "just to log some metrics." At low traffic it added 40ms. Under load, it held a connection long enough to starve the pool, and every unrelated endpoint that touched the same database went down with it.

N+1 Queries You Never Noticed

An N+1 query pattern — fetching a list, then issuing a separate query per item to load its details — is invisible with 10 items in a list. It's still mostly fine with 100. Somewhere around a few thousand concurrent users hitting that same endpoint, it stops being a code-quality nitpick and starts being an outage, because you've quietly multiplied your database load by the size of every list on the page.

Missing Indexes That Only Bite Under Load

A full table scan on a few thousand rows is fast enough that nobody notices in staging. The same scan on ten million rows, run concurrently by hundreds of requests a second, turns into lock contention and CPU saturation on the database host. The fix is usually a five-minute index migration — the hard part is that nobody looks for it until production is already on fire.

⚡ Caching Layers: Buy Yourself Time, Don't Hide Bugs

Caching is the highest-leverage fix available once the database starts to strain, but it has to be deliberate. A cache that's bolted on in a panic during an incident usually just moves the failure mode somewhere less visible.

- **Read-through caching for hot data:** user profiles, product catalogs, configuration — anything read far more often than it's written is a good candidate for a cache in front of the database, with a sane TTL as the safety net.
- **CDN edge caching for anything static or near-static:** images, public API responses, rendered pages — push them as close to the user as possible so they never reach your origin at all.
- **The invalidation problem is the real cost:** a stale cache that serves an old price or an old permission for thirty seconds can be a worse incident than the slow query it was meant to hide. Design invalidation before you design the cache, not after.

Stateless Services and Horizontal Scaling

The single most common architectural wall I've seen teams hit is session state living on one server. It starts innocently — a login session stored in an in-process map because it's fast and it works in local dev. It keeps working fine right up until you need a second server behind a load balancer, and suddenly half your users get logged out depending on which instance they land on.

Once every request-handling service is stateless — session data in a shared store, no server-local caches that matter for correctness, no in-memory queues that would lose work on restart — horizontal scaling becomes a capacity decision instead of an architecture problem. You add servers behind the load balancer and traffic distributes. That's the whole point of doing the boring work early.

Queues and Async Processing

Synchronous request handling is the first thing to buckle under a traffic spike, because every request holds a thread, a connection, and often a database transaction for as long as the slowest thing it's waiting on. Anything that doesn't need to happen before you respond to the user — sending an email, resizing an image, calling a third-party API that's occasionally slow — belongs in a queue, not in the request path.

The benefit isn't just speed. A queue decouples the rate work arrives from the rate you can process it. During a spike, the queue absorbs the burst and your workers catch up

over the next few minutes instead of your web servers timing out in real time while users watch a spinner.

Observability Before You Need It

You cannot fix what you cannot see, and the worst time to add tracing, structured logging, and per-endpoint latency metrics is during an active incident. By the time things are on fire, you need answers in minutes, not the twenty minutes it takes to instrument a system that's never had visibility before.

The minimum I want in place before traffic grows: request-level tracing that survives across service boundaries, database query timing broken out by endpoint (not just an aggregate), and alerting on connection pool utilization and queue depth — not just error rate. Error rate tells you something already broke. Pool utilization and queue depth tell you something is about to.

Cheap Now, Expensive Later

Some decisions cost almost nothing to make correctly on day one and cost weeks of migration work to fix after you have real traffic and real data. Others are genuinely fine to defer. Knowing which is which is most of what "designing for scale" actually means in practice.

Decide correctly now

- A scalable, non-sequential ID scheme for anything that will be sharded or replicated later
- Idempotent APIs — retries under load should never double-charge or double-send
- No server-local state that anything correctness-critical depends on
- Indexes on anything you query by in a hot path

Safe to defer

- Splitting into microservices before you have a team-scaling reason to
- An exotic database chosen for scale you don't have yet
- Multi-region active-active — most products need multi-AZ, not multi-region

- Custom caching infrastructure before a managed cache has proven insufficient

The Honest Takeaway

Scale problems are mostly about sequencing which fire to put out first, not some grand upfront architecture nobody could have gotten right on day one. The database strains before anything else notices. Caching buys time if you design invalidation deliberately. Statelessness is what makes adding capacity a non-event instead of a rewrite. And observability has to exist before the incident, because you don't get to add it during one.

None of this requires predicting your future traffic correctly. It requires not painting yourself into the handful of corners that are genuinely expensive to get out of later — and being honest that everything else can wait until you actually need it.