FREE DIAGNOSTIC CHECKLIST

The Slow Web App Diagnostic

Find which layer is stealing your seconds: frontend, backend, or database. One afternoon, no APM required.

My favorite interview question for senior engineers is one sentence: "The web app you built is slow. What do you do?" Weak answers jump straight to a favorite fix (add an index, add a cache, scale up). Strong answers start with triage. This checklist is the strong answer, written down.

Step 0: Triage. Five questions before touching anything

The answers route you to the right section and save you days of guessing.

1. **Always slow, or slow under load?** Always slow = a code or query problem, go layer by layer below. Slow only under load = a resource or pool limit, start at Backend and Database server checks.
2. **Did it get slow as data grew?** Year-old app, growing tables, slowly degrading = almost always the database: queries and indexes that were fine at 10k rows and are not fine at 10M.
3. **Did it get slow after a deploy?** Diff the release. The bottleneck is in the change, not in the architecture.
4. **Slow for everyone, or some users?** Some users = look at their data shape (the customer with 100x more records), their region (latency, no CDN), or their device (frontend weight).
5. **One page, or everything?** One page = trace that page. Everything = shared infrastructure: database server, connection pools, instance size.

> **Measure before fixing.** Percentiles, not averages: an average of 300ms can hide a p99 of 8 seconds, and your noisiest customers live in the p99. Trace ONE slow request across the stack and write down where the milliseconds go: browser → network → backend → database → third parties. The split tells you which section below to open.

WHERE THE TRACE SENDS YOU

Part 1: Frontend. Cheapest to rule out, check it first

The fastest split: TTFB (time to first byte) vs everything after. TTFB slow = backend problem, skip to Part 3. TTFB fine but page slow = frontend, stay here.

- [ ] **TTFB under ~500ms?** If yes, the backend is mostly innocent for this page.
- [ ] **JS bundle size.** Multi-MB bundles take seconds to parse on mid-range phones. Code-split, drop dead dependencies.
- [ ] **Images.** Still the #1 page-weight killer: serve WebP/AVIF, size to the container, lazy-load below the fold.
- [ ] **CDN in front of static assets?** Assets served from one region make every other continent pay round-trip tax.
- [ ] **Third-party scripts.** Analytics, chat widgets, session recorders. Measure with them disabled; the difference is regularly shocking.
- [ ] **Render-blocking resources.** CSS and sync scripts in head that delay first paint.
- [ ] **Run Core Web Vitals (PageSpeed Insights).** Free, one minute, and Google grades the exact metrics your users feel: LCP, INP, CLS.

Part 2: Network. The layer everyone forgets exists

- [ ] **Payload sizes.** API responses with 200 fields when the page renders 6. Slim the response or paginate.
- [ ] **Compression on?** gzip/brotli for JSON and HTML. Free 5-10x on text payloads, still commonly off.
- [ ] **Cache headers.** Static assets re-downloaded on every visit because nobody set Cache-Control.
- [ ] **Chatty APIs.** 25 sequential requests to render one page. Batch them, or fetch in parallel.
- [ ] **Geographic latency.** Server in us-east-1, customers in Australia: 200ms+ per round trip before any code runs.

Part 3: Backend. Server first, code second

Check the server before profiling the code. A perfect app on a starved box is still slow.

- [ ] **CPU, memory, IO on the app host.** Sustained CPU >80%, swapping, or IO-wait = resource problem. Before scaling up, ask WHY it's hot (the answer is often in Part 4).
- [ ] **Burstable instances (t3/t4g) out of CPU credits.** The classic invisible killer: app fast all morning, crawling by afternoon. Check the credit balance graph.
- [ ] **Thread pool / worker exhaustion.** Requests queue before your code even runs. Check pool utilization and queue depth.
- [ ] **Connection pool to the database.** 200 app threads sharing 10 connections = 190 threads waiting. Pool wait time is the metric.
- [ ] **Then profile the app.** Flame graph on one slow endpoint. Look for: work redone every request that should be cached, JSON serialization of huge objects, sync work that belongs in a queue.
- [ ] **GC pauses (JVM/.NET/Go).** Latency spikes at regular intervals with clean CPU = collect GC logs.

Part 4: Database. Where most of the seconds usually are

- [ ] **DB server params first: IO, memory, CPU.** IO-bound = working set no longer fits in memory; that's a config or query problem before it's a hardware problem.
- [ ] **Slow query log on.** Top 3 slowest queries by total time. Fix those, ignore the rest for now.
- [ ] **EXPLAIN the top 3.** Sequential scans on big tables = missing or unused index.
- [ ] **Indexes: more is not better.** Don't index every column "just in case." Every index taxes every INSERT/UPDATE, bloats storage, and confuses the planner. Index what your WHERE/JOIN/ORDER BY actually use, drop the rest.
- [ ] **Correct column types.** Dates as strings, UUIDs as text, numbers as varchar: every comparison pays a conversion, and indexes work badly or not at all.
- [ ] **N+1 queries.** One query for the list, then one per row. The ORM hides it; the query log doesn't. 1 + 200 fast queries is still slow.
- [ ] **Lock contention.** Fast queries that intermittently take seconds = they're waiting on locks, not working.

Part 5: Third-party calls. Other people's latency becomes yours

- [ ] **Sequential external calls.** Three vendor APIs at 300ms each, one after another = 900ms floor. In parallel = 300ms.
- [ ] **No timeouts.** One slow vendor freezes your whole request, then your whole thread pool. Set timeouts everywhere, decide a fallback.
- [ ] **External calls inside loops.** The N+1 problem, but over the internet. Batch endpoints exist; use them.
- [ ] **No caching of stable data.** Exchange rates, geo lookups, settings fetched fresh on every request.

The traps: mistakes that make it worse

⚠ **Scaling up instead of diagnosing.** Doubling the instance size to hide a missing index = paying a monthly subscription for a one-time fix nobody did.

⚠ **Caching as a band-aid.** A cache over a broken query means you now have two problems, and the second one shows up as stale-data bugs.

⚠ **Index on every column.** Writes get slower, the planner gets dumber, the bill gets bigger.

⚠ **Trusting averages.** Average latency is marketing. p95/p99 is engineering.

⚠ **Optimizing without a baseline.** If you didn't measure before, you can't prove the fix worked. Measure, change ONE thing, measure again.

Want a second pair of eyes?

I do this diagnostic on real systems: a 30-minute call, your metrics on screen, and you leave knowing which layer to fix first.

Book a free 30-minute call

Viktar Patotski · Software Engineer & Architect, 20+ years · **patotski.com**