FREE ROADMAP · FOR DEVELOPERS · 2026

System design fundamentals.

The 10 concepts to learn first, in order, and a 4-week plan to actually learn them.

> **Why people struggle:** they start with the advanced stuff (microservices, sharding, event streaming) before they understand the basics. In an interview it shows within minutes: you can draw boxes, but you can't explain why they're there. In production it gets expensive. In ITIC's 2024 survey, **97% of large enterprises** said a single hour of downtime costs them **over $100,000**.

Inside:

① The 10 fundamentals, each with what to learn and the question to ask yourself
② A 4-week learning plan with 2 practice systems to design
③ A 6-step framework for any system design interview



Afeez Badmos
Principal Technical Lead · Global McLien Software Solutions

Downtime figure: ITIC 2024 Hourly Cost of Downtime Survey (organizations with more than 1,000 employees).

Learn them in this order

TICK AS YOU GO

#	CONCEPT · WHAT TO LEARN	ASK YOURSELF	✓
01	**Client–server architecture** How a request travels: client → DNS → server → database and back. Stateless vs stateful servers.	Where could this request fail, and how would I know?	☐
02	**HTTP & APIs** Methods, status codes, headers, REST resource design, pagination, idempotency, versioning.	What happens if the client retries this call twice?	☐
03	**Databases: SQL vs NoSQL** Relational modelling, indexes, transactions (ACID) vs document/key-value stores; read vs write patterns.	What are my queries, and does my data have relations?	☐
04	**Caching** Cache-aside, TTLs, invalidation, CDN vs in-memory (e.g. Redis); hit rate as the metric that matters.	What's read often, changes rarely, and can be a little stale?	☐
05	**Load balancers** Round-robin vs least-connections, health checks, L4 vs L7, sticky sessions and why to avoid them.	What happens when one server dies mid-request?	☐
06	**Scalability: vertical vs horizontal** Bigger machine vs more machines; read replicas; partitioning/sharding as the last step, not the first.	What breaks first at 10× traffic?	☐
07	**Monolith vs microservices** Start with a well-structured monolith; split when team size or deploy pain demands it.	What problem would splitting this solve, today?	☐
08	**Message queues** Producers, consumers, async work, retries, dead-letter queues, at-least-once delivery.	Does the user need to wait for this work to finish?	☐
09	**CAP theorem & consistency** During a network partition you choose consistency or availability; strong vs eventual consistency.	Is showing slightly old data OK here, or never?	☐
10	**Architectural trade-offs (the kicker)** Weigh performance, availability, cost and complexity for *your* requirements instead of copying a popular architecture.	What am I giving up with this choice, and is that OK?	☐

Rule of thumb: if you can't explain concepts 1–5 out loud with a simple diagram, skip microservices and sharding for now. They build on everything above them.

4-week learning plan ~5 H / WEEK

WEEK 1 **Foundations: #1–#3.** Build a tiny API with one SQL table. Read your own request logs end to end and break it on purpose (wrong status codes, slow query).

WEEK 2 **Speed & traffic: #4–#6.** Add a cache to the slow endpoint and measure before/after. Put two instances behind a load balancer and kill one.

WEEK 3 **Shape & flow: #7–#9.** Move one slow task (e.g. sending email) to a queue. Write down what happens if the consumer is down for an hour.

WEEK 4 **Trade-offs: #10.** Design both practice systems below on paper, then list 3 trade-offs you made and what you'd change at 100× users.

Practice systems DESIGN ON PAPER

- [ ] **URL shortener:** API design, ID generation, read-heavy caching, database choice, redirects at scale.
- [ ] **Chat app:** real-time connections, message storage, queues for notifications, ordering and consistency.

Any system design interview, in 6 steps 45 MIN

1 · Requirements
Functional and non-functional: users, scale, latency, availability.

2 · Estimates
Rough reads/writes per second and storage. Show your maths.

3 · API
The key endpoints and what they return.

4 · High-level design
Client, load balancer, services, database, cache.

5 · Deep dive
The hardest part: data model, bottleneck, or consistency.

6 · Trade-offs
What you chose, what you gave up, what you'd do at 10×.

BUILDING SOMETHING REAL?

Get an architecture review before you scale.

Global McLien Software Solutions designs and reviews web apps, mobile apps, SaaS platforms and cloud infrastructure, so your system holds up on launch day.

globalmcliensoftwaresolutions.com →

Sources: ITIC 2024 Hourly Cost of Downtime Survey: itic-corp.com · CAP theorem: Gilbert & Lynch, "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services" (ACM SIGACT News, 2002).