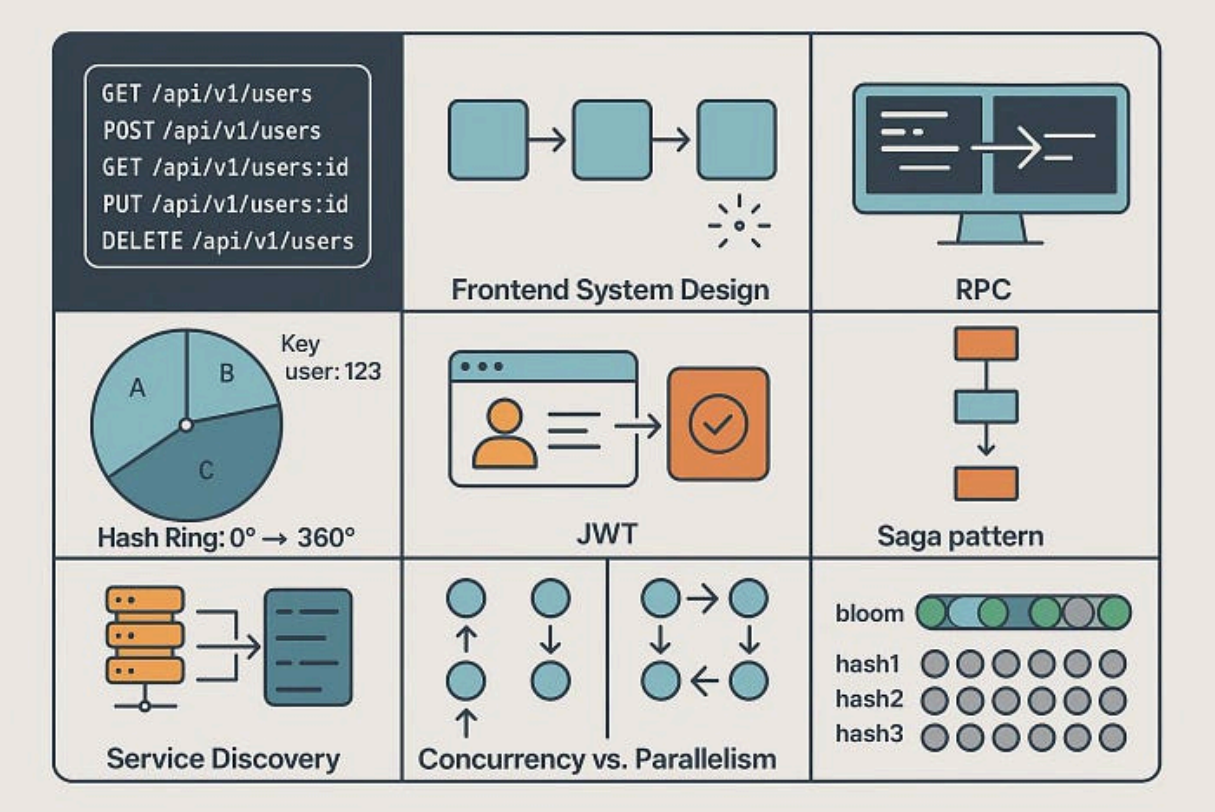

If You Don't Know These 16 System Design Concepts, You're Not Senior Yet

After interviewing at multiple tech companies and reviewing hundreds of system design submissions, I've noticed a pattern. Junior...

The Latency Gambler

Follow

a11y-light · December 16, 2025 (Updated: December 16, 2025) · Free: No

After interviewing at multiple tech companies and reviewing hundreds of system design submissions, I've noticed a pattern. Junior developers focus on coding. Mid-level developers focus on architecture. But senior developers? They think in systems.

level up.

1. API Design Best Practices

Good APIs are like good conversations clear, predictable, and respectful of everyone's time. RESTful conventions matter: use nouns for resources, HTTP methods for actions, and proper status codes. Version your APIs from day one because breaking changes will happen.

Copy

```
// Good API design
GET    /api/v1/users
POST   /api/v1/users
GET    /api/v1/users/:id
PUT    /api/v1/users/:id
DELETE /api/v1/users/:id
```

Consistency in naming, error handling, and response formats isn't optional , it's what makes your API maintainable at scale.

2. Frontend System Design 101

Frontend architecture isn't just about React vs Vue. It's about state management, code splitting, caching strategies, and performance budgets. Consider how your components communicate, where state lives, and how you'll handle real-time updates. A well-designed frontend anticipates network failures and provides meaningful loading states.

3. How RPC Works

action-oriented. gRPC, using Protocol Buffers, offers type safety and impressive performance. The tradeoff? Less human-readable than JSON, and tighter coupling between services.

4. How Consistent Hashing Works

When you're distributing data across multiple servers, naive hashing breaks during scale-up or scale-down. Consistent hashing solves this by mapping both data and servers onto a ring. Adding or removing a server only affects its immediate neighbors, minimizing data movement. This is how Cassandra, DynamoDB, and most distributed caches maintain efficiency.

Copy

```
Hash Ring: 0° → 360°
Server A: 45°
Server B: 180°
Server C: 270°

Key "user:123" → hash(270°) → routes to Server C
```

5. How JWT Works

JSON Web Tokens provide stateless authentication. The server signs a token containing user claims, the client stores it, and subsequent requests include this token. No database lookup needed for validation , just verify the signature. The catch? You can't revoke JWTs before expiration without additional infrastructure like token blacklists.

6. System Design 101

components, and identify bottlenecks. Think about CAP theorem trade-offs, single points of failure, and how you'll monitor the system in production. Design isn't about perfect solutions , it's about informed trade-offs.

7. How Saga Design Pattern Works

Distributed transactions are hard. The Saga pattern breaks long-running transactions into smaller, local transactions with compensating actions for rollback. If booking a flight succeeds but the hotel fails, you trigger a compensation to cancel the flight. This maintains eventual consistency without distributed locks.

Choreography approach: Each service publishes events that trigger the next step. **Orchestration approach:** A central coordinator directs the workflow.

8. How Service Discovery Works

In dynamic cloud environments, service instances constantly change. Service discovery solves the "how do I find other services?" problem. Client-side discovery (like Netflix Eureka) has clients query a registry. Server-side discovery (like Kubernetes) routes through a load balancer that knows the current topology.

9. Concurrency vs Parallelism

Concurrency is about dealing with multiple things at once. Parallelism is about doing multiple things at once. A single-core CPU can run concurrent programs by rapidly switching contexts, but only multi-core systems achieve true parallelism. Understanding

10. API Versioning: A Deep Dive

Your API will evolve. Version it in URLs (`/v1/users`), headers (`Accept: application/vnd.api+json;version=1`), or query parameters. URL versioning is most transparent but pollutes your routes. Header versioning is cleaner but less discoverable. Support multiple versions during deprecation windows, communicate changes clearly, and never break existing clients without warning.

11. Modular Monolith Explained

Not every system needs microservices from day one. Modular monoliths organize code into independent modules with clear boundaries, but deploy as a single unit. You get microservices' organizational benefits without the operational overhead. When growth demands it, well-defined modules extract cleanly into separate services.

12. How Bloom Filters Work

Need to check if an element exists in a massive dataset without loading everything into memory? Bloom filters use multiple hash functions to mark bits in an array. They guarantee no false negatives but allow false positives. Perfect for cache layers: "This URL definitely hasn't been crawled" or "This email might be in our spam list."

Copy

```
# Simplified concept
bloom = [0] * 100
hash1("user@email.com") → 23 → bloom[23] = 1
```

13. How Idempotent APIs Work

Network requests can fail, retry, or duplicate. Idempotent operations produce the same result regardless of how many times they execute. GET and PUT are naturally idempotent. POST isn't submitting the same order twice creates two orders. Use idempotency keys to make non-idempotent operations safe for retries.

14. How Databases Keep Passwords Securely

Never store plaintext passwords. Hash them with bcrypt, scrypt, or Argon2 algorithms designed to be computationally expensive. Salt each password uniquely to prevent rainbow table attacks. When users log in, hash their input and compare hashes. Even if your database leaks, attackers face massive computational barriers.

15. Cloud Architecture Pitfalls

The cloud isn't just someone else's computer , it's a different paradigm. Common pitfalls include assuming unlimited resources (then hitting account limits), ignoring egress costs (data transfer out is expensive), treating instances as pets instead of cattle, and forgetting that availability zones fail independently. Design for failure, automate everything, and monitor your bill.

16. Distributed Systems: A Deep Dive

This is where everything converges. Distributed systems introduce partial failures, network latency, clock synchronization issues, and data consistency challenges. Master concepts like consensus algorithms (Raft, Paxos), eventual consistency, vector clocks, and distributed tracing. Understand that you can't have perfect

The Path Forward

These concepts aren't academic exercises. They represent real problems solved by real engineers building systems at scale. You don't need to master all sixteen tomorrow, but you should recognize them when they appear and know where to dig deeper.

Senior engineering isn't about memorizing patterns , it's about understanding why they exist, when to apply them, and what trade-offs they introduce. Start with problems you're facing today, then learn the concepts that solve them.

The systems you build tomorrow depend on the fundamentals you learn today. Which concept will you explore first?

#system-design-concepts #system-design-interview #software-engineering #programming #coding-interviews