HOW TO APPROACH A SYSTEM DESIGN INTERVIEW QUESTION

How to tackle a system design interview question.

The system design interview is an **open-ended conversation**. You are expected to lead it.

Part 1:

- Do not panic!
- Don't think like a coder. Think like a Tech Lead.
    - During the interview, you'll spend an hour playing the role of a Tech Lead, so just pretend that the interviewer is a junior engineer who will be implementing your design.
- There are no optimal solutions in system design interviews.
- It's your responsibility to leave breadcrumbs for the interviewer to go where you want them to go. That way you have them walk you down the road where you are at your best
- You do not need to display deep expertise in the given problem domain.
- Interviewers want to see that you have a broad, base-level understanding of system design fundamentals.
- Interviewers want to engage you in a back-and-forth conversation about problem constraints and parameters, so avoid making assumptions about the prompt.
- Interviewers are not looking for specific answers with ironclad certainty. They want to see well-reasoned, qualified decisions based on engineering trade-offs.

- Interviewers are not looking for a predefined path from the beginning to end of the problem. They want to see the unique direction your experience and decisions take them.
- Interviewers seek a holistic view of a system and its users.
- Communicate honestly about what you know and what you don't.
- For senior role, it's a good sign if you direct more of the interview
- A common failure point occurs when candidates don't make decisions
- Not being familiar with specific databases or other components is fine. Be smart and don't say brand names just for the sake of saying them.
    - "I'm going to use Cassandra ..." unless you are VERY familiar with that, because the next question will be: "Why Cassandra and not some_other_db?"
    - "I will use Kafka ..." unless you're prepared to explain how Kafka works. Don't say "I will use Kafka" unless you are prepared to talk about other types of queues, because they may ask you: "Oh, Kafka, interesting choice. Why that instead of [some other queue]?"

Part 2:

- There's no right way to design a system.
- If the interviewer interrupts you, it's probably because you're going off track.
- It's fine if the interviewer asks you questions, but it's a bad sign if the interviewer starts telling you how to do things.
- It's more important to cover everything broadly than it is to explain every small thing in detail.
- Whatever decision you make, explain why. In a system design interview, why is more important than what. For anything you say, be prepared to explain why.

- Mock interviews with different types of interviewers are the best solution we've found to refining your communication skills, or working with a dedicated coach who can get to know you (and your areas of expertise and improvement) very well.
- There is no strictly wrong answer for which database to use, as long as you can justify yourself and demonstrate an understanding of the alternative options.

Part 3:

- You should start with the functional requirements first—that is, the core product features and use cases that the system needs to support.
- What makes these questions difficult or complicated is not the fact that they are inherently complicated. It's because the interviewer intentionally chooses to withhold information. Information that you can only get if you ask the right questions.
- Treat the system as a black box. No thinking about design, implementation, or pretty much anything technical. The sole goal of this first step is to specify what needs to be built. Not how. Not the scale. Focus on the "what."
- Consider mutability. Can tweets be edited after they're published? Can tweets be deleted?
    - It might sound like a small detail at first, but mutability can limit our ability to use caching in our design
- Your interviewer cares less about your decisions, and more about whether you are able to talk about the trade-offs (positives and negatives) of your decisions

- We measure availability by the percentage of the time the system is up and running. A common goal is to aim for five nines, i.e., 99.999% availability—that's less than 6 minutes of downtime a year.
- Whenever there is user-generated code execution involved (aka low trust code), running it in isolation should be a non-functional security requirement.
- Want speed, use a cache. Want availability, put in some redundancy. Want fast reads to db, use a read replica. Want fast writes to db, use sharding.
- Consider using caching when all three of these are true:
    - Computing the result is costly
    - Once computed, the result tends to not change very often (or at all)
    - The objects we are caching are read often

Part 4:

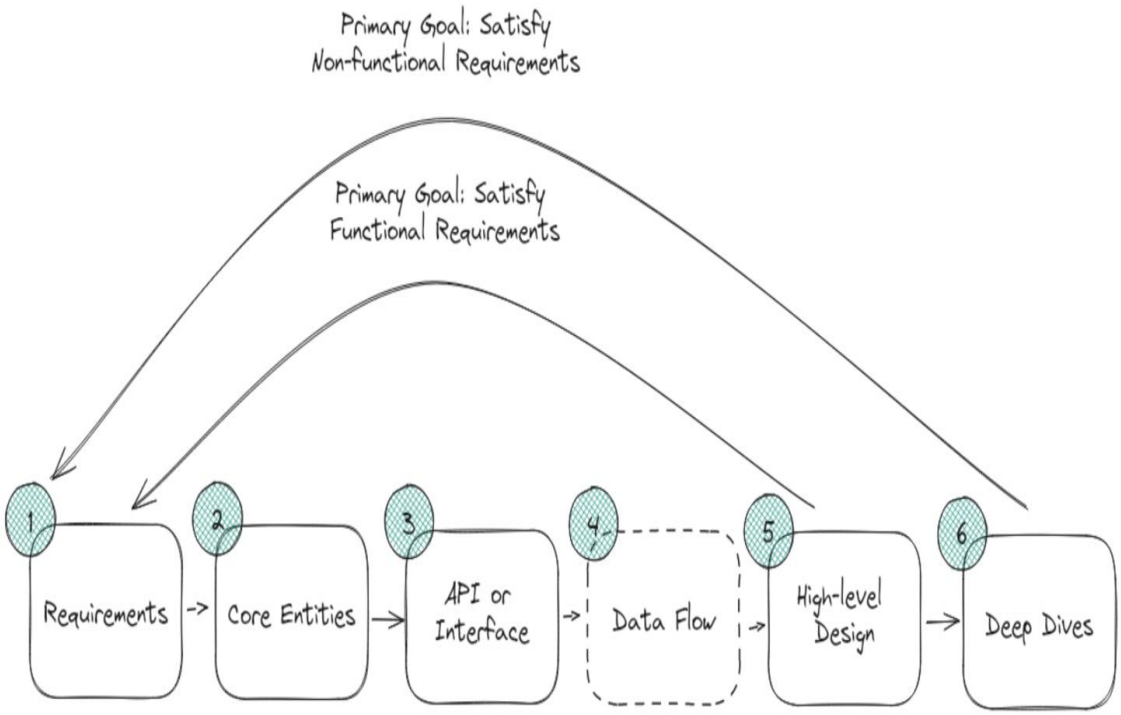
- System Design problems are almost always easier in the immutable case (compared to the mutable one).
- How to Get Yourself Unstuck – Tip #1 - To simplify the problem, think about the data in the problem as immutable. You can choose to add mutability back in later after you have an initial working design.
- How to Get Yourself Unstuck – Tip #2 - Ask your interviewer: "Are there any important requirements you have in mind that I've overlooked?"
- How to Get Yourself Unstuck – Tip #3 - Ask your interviewer if they'd like to see some calculations before jumping in and starting them—you might be able to skip these entirely if the interviewer doesn't care about it!
- How to Get Yourself Unstuck – Tip #4 - Follow these rough guides to get the basic estimates for any system

- Storage Estimation: Storage = daily data used by 1 user * DAU count * length of time to store data
- Bandwidth Estimation: Bandwidth per second = (daily data used by 1 user * DAU count) / total seconds in a day

- How to Get Yourself Unstuck – Tip #5 “Would it be fine if the data in my system was occasionally wrong for a split second or so?”
  - If the answer is **yes**, then you probably want eventual consistency.
  - If the answer is **no**, then you’re looking for a strong consistency called linearizability.

System Design interview requirements

Do not jump right in to give a solution. Slow down. Think deeply and ask questions to clarify requirements and assumptions. This is extremely important.



- Requirements (3-5 min)
    - Functional
        - Start with the customer and work backwards
        - **Who** will use the system
        - **How** the system will be used
        - Choosing just the top 3 features
        - Adding features that are out of scope is a nice to have, it shows product thinking
        - Ask your interviewer: “Are there any important requirements you have in mind that I’ve overlooked?”

- Non functional (use **why** you used such NFR)
    - **High availability** system uptime in percentage
        - 99% ⇒ 3.65 days of downtime per year
        - 99.9% ⇒ 8.76 hours of downtime per year
        - 99.99% ⇒52 minutes 35 seconds of downtime per year
        - 99.999% ⇒5 minutes 15 seconds of downtime per year
        - 99.9999% ⇒31 seconds of downtime per year
    - **Scalability** the property of a system to handle a growing load
        - Vertical
        - Horizontal
        - Elastic — ability to scale up and down
        - Auto-scaling
    - **Performance** in terms of latency and throughput
    - **Rezilience** is how quickly the system can recover from failures
    - **Durability** - data should not be lost
        - Backups (full, differential, incremental)
        - RAID
        - Replication
    - **Consistency** - data should not be corrupted
        - Strong consistency
        - Eventual consistency
        - Causal Consistency — correct order of events
    - **Maintainability**

        - Failure modes and mitigations
        - Monitoring
        - Testing
        - Deployment
    - **Security**
        - Authentication and Authorization
        - DDOS protection
        - SQL injection
        - Data protection in transit and storage
    - **Cost**
        - Engineering cost
        - Maintenance cost
        - Resource cost

- Core entities (2 min)
    - Example: User, Post, Comment, Like, Event, Ticker, etc.

- API design (5–7 min)
    - Putting sensitive information like userId in the request body is a security risk
    - It's ok to have simple APIs from the start that you evolve as your design progresses
    - Public endpoints
        - CRUD

    - Private endpoints

- High level design (10–15 min)
    - Start with something simple (Focus on a relatively simple design that meets the core functional requirements, and then layer on complexity to satisfy the non-functional requirements in your deep dives section)
        - Single service solution
        - Design monolith service and then break it into microservices
    - APIs for Read/Write scenarios for crucial components
    - Simple database schema next to the database components, according to the **core entities**
    - High level design for a Read/Write scenario
        - High level design for Read heavy scenario
        - High level design for Write heavy scenario

- Deep dives (15-20 min)
    - Scaling the high level design
    - Scaling individual components:
        - Availability, Consistency and Scale story for each component
        - Consistency and availability patterns
    - Think about the following components, how they would fit in and how it would help:
        - Client (Mobile, Browser)
        - DNS

- CDN (Push vs Pull)
- Load Balancers (Active-Passive, Active-Active, Layer 4, Layer 7)
- Reverse Proxy
- Blob Storage
- Application layer scaling (Microservices, Service Discovery, N tier)
- Database:
    - **SQL**
        - Replication, Sharding, Indexing, Denormalization, SQL Tuning
    - **NoSQL**
        - Types: Key-Value, Wide-Column, Graph, Document, Geospatial, Vector, Full-Text scanning
        - Read heavy: MongoDB, Couchbase
        - Write heavy: Cassandra, ScyllaDB
    - **Cold/Hot storage**
- Cache:
    - Client caching, CDN caching, Webserver caching, Database caching, Application caching
    - Eviction policies:
        - LRU, LFU
    - Types:
        - Cache aside, Write through, Write behind
- Queue:

            - Message queues
            - Task queues
            - Back pressure
        - Communication:
            - HTTP/S, TCP, UDP, RPC, Web Sockets
    - Logging, Metrics and Automation

- Estimations (3–5 min) (ask interviewer if needed)
    - Latency/Throughput expectations
    - QPS (Queries Per Second) Read/Write ratio
    - Total/Daily active users
        - Traffic estimates
            - Write (QPS, Volume of data)
            - Read (QPS, Volume of data)
        - Storage estimates
        - Memory estimates
            - If we are using a cache, what is the kind of data we want to store in cache
            - How much RAM and how many machines do we need for us to achieve this?
            - Amount of data you want to store in disk/ssd
- Follow-up (2–3 min)
    - Bottlenecks
    - SPOF

- Latency (use other protocols, cache, multi-threading, faster algorithms, compression, cdn)
- Security
- Cost
- Error cases (server failure, network loss, etc.) and how to handle them
- Monitoring
- How to scale the system to the next level

Availability	Scalability	Performance	Durability	Consistency
System uptime, the percentage of time the system has been working and available.	The property of a system to handle a growing load.	The time required to process something and/or the rate at which something is processed.	Once data is successfully submitted to the system, it is not lost.	Consistency of data across distributed copies.
• high availability • SLO and SLA • fault tolerance • resilience • chaos engineering • reliability	• vertical and horizontal scaling • elasticity • autoscaling	• latency • percentiles • throughput • bandwidth	• backup • RAID • replication • data corruption and checksum	• strong consistency • weak consistency • consistency model • linearizability • CAP • eventual consistency • monotonic reads • read-your-writes • consistent prefix reads

Maintainability	Security	Cost
The ease with which a product can be maintained.	Degree to which the system protects against threats.	How to design systems with the most effective use of resources.
• failure modes and mitigations • monitoring • testing • deployment	• CIA triad • identity and permissions management • infrastructure protection • data protection	• engineering cost • maintenance cost • hardware cost • software cost

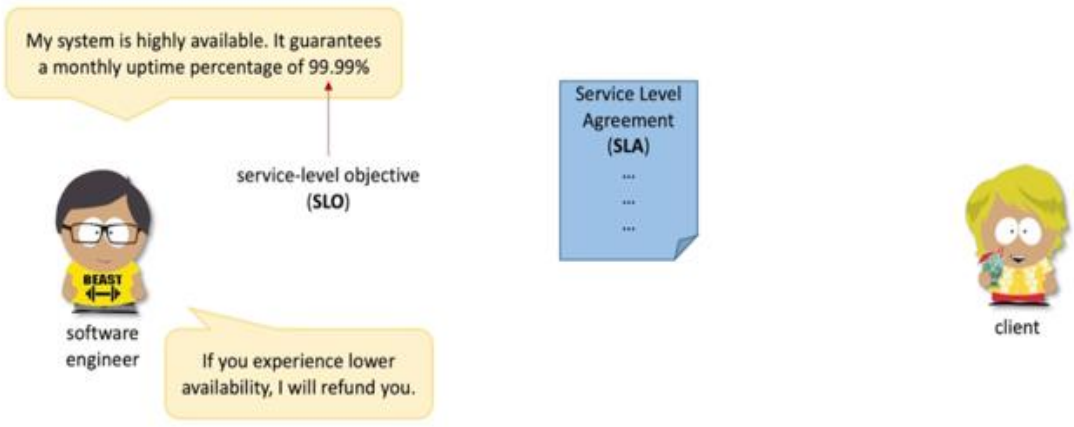
Availability

Non-functional requirements – High availability

design principles behind high availability

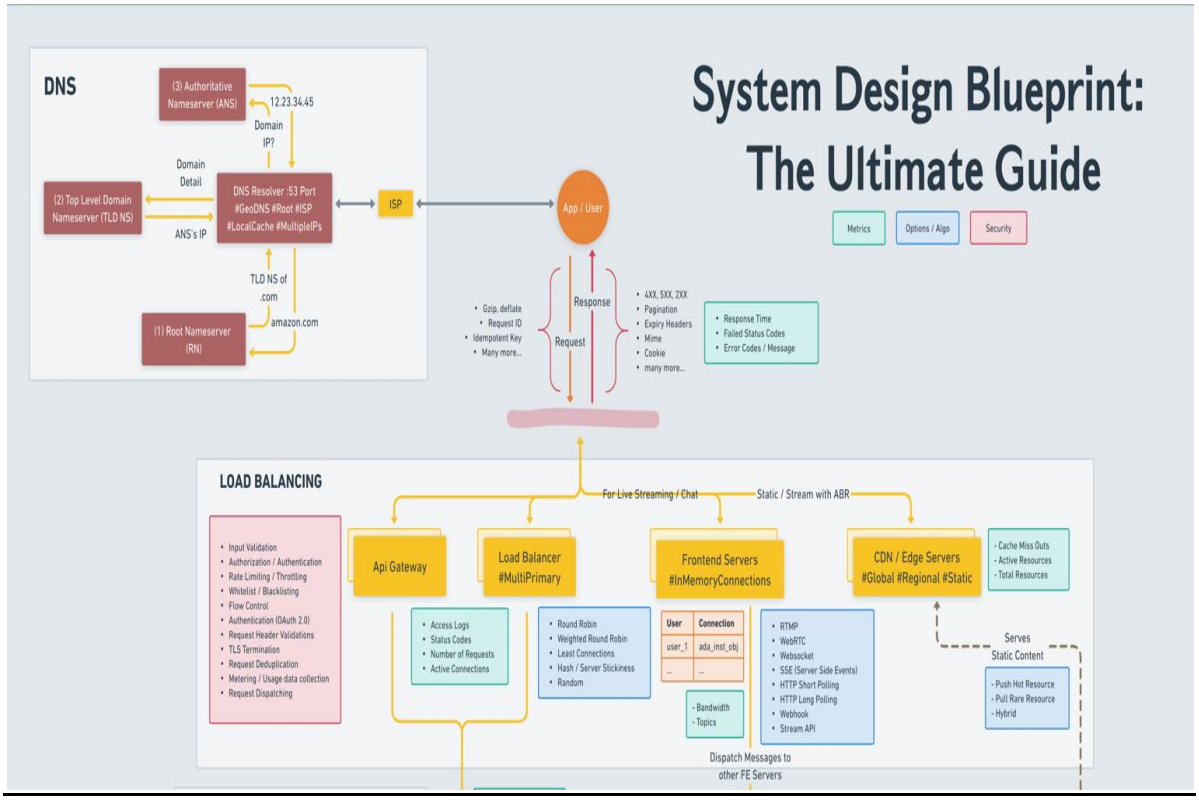
- build redundancy to eliminate single points of failure
  (regions, availability zones, fallback, data replication, high availability pair, ...)
- switch from one server to another without losing data
  (DNS, load balancing, reverse proxy, API gateway, peer discovery, service discovery, ...)
- protect the system from atypical client behavior
  (load shedding, rate limiting, shuffle sharding, cell-based architecture , ...)
- protect the system from failures and performance degradation of its dependencies
  (timeouts, circuit breaker, bulkhead, retries, idempotency, ...)
- detect failures as they occur
  (monitoring, ...)

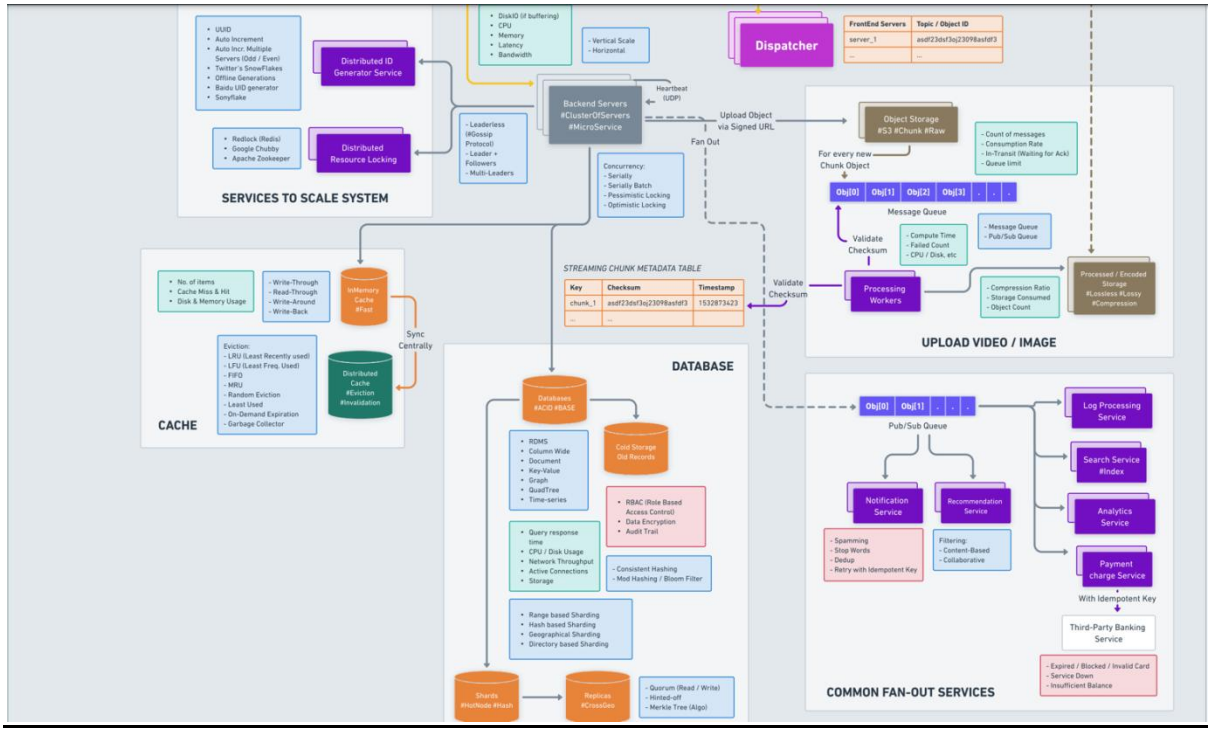
Non-functional requirements – High availability



Availability %	Downtime per day	Downtime per year
99%	14.40 minutes	3.65 days
99.9%	1.44 minutes	8.77 hours
99.99%	8.64 seconds	52.60 minutes
99.999%	864.00 milliseconds	5.26 minutes
99.9999%	86.40 milliseconds	31.56 seconds

System Design Blueprint: The Ultimate Guide





High Load Applications Architecture

Problems

- Single point of failure - SPOF is a part of a system that, if it fails, will stop the entire system from working. SPOFs are undesirable in any system with a goal of high availability or reliability.
- Bottlenecks - Bottleneck occurs when the capacity of an application or a computer system is severely limited by a single component. It has lowest throughput of all parts of the transaction path. **System works as fast as the slowest part of the system**.

Main principles

- Reliability
- Scalability
- Efficiency
- Performance

Simplicity is a key

- It is better to have a bunch of simple components in the system.
- Simple things are easier to understand, analyze, improve, replace.
- Look at SOLID. S there stands exactly for that - Single Responsibility Principle. Each component in the system should serve one and only one goal.

Proper approach for building High Load Applications

- Always keep in mind what is your main goal
- Start with simplest solution (**Simple is not weak**)
- Continuously look for bottlenecks and SPOFs - improve first and avoid second
- Split complex things into simple ones