If I had a system design interview tomorrow at Amazon or Microsoft, this is exactly how I’d approach it.

(This framework helped me crack 3 FAANG+ companies in the past.)

Most engineers don’t fail system design interviews because of lack of knowledge.

They fail because they lack structure.

You don’t need to memorize Netflix’s architecture or read 200 blog posts.

You need a framework that works every single time, one that helps you think clearly, communicate confidently, and make smart trade-offs.

Here’s how I personally approach every system design interview and why it’s helped me stay consistent across the board:

1/ Clarify the Problem Statement (Don’t Jump the Gun)

Before drawing any diagram or suggesting a database, I start by asking:

- Are we designing an entire system or just one core feature?

- Who are the users and what are their usage patterns?

- Are we optimizing for latency, throughput, or cost?

- Are we dealing with real-time updates or batch processing?

🔁 Most candidates assume everything.

I never assume. I clarify, scope, and set clear boundaries upfront.

Example: “Design A Content Streaming Platform”   
→ Are we focused on uploads? Live streaming? Recommendations?

Each one has a very different architecture.-

2/ Define Functional and Non-Functional Requirements Clearly

Functional: What the system should do

(e.g., user login, video upload, real-time messaging)

Non-Functional: How the system should behave

(e.g., 99.99% uptime, handle 100K QPS, global access, sub-100ms latency)

✅ I also ask: “What’s the hardest part technically?”

This tells me where to focus deeper design.

3/ Estimate Scale and Do Capacity Planning

This is where most people mess up. Throwing “millions of users” means nothing unless you translate it into numbers.

- Daily Active Users (DAU)?

- Requests per second (RPS)?

- Read/write ratio?

- Data stored per user per day?

- Peak traffic hours?

Example: If we store 50MB per user and we have 10M users → that’s 500 TB of storage.

If each user sends 10 requests per day → 100M requests daily = 1157 RPS.

Without realistic estimates, your system will either crash in prod or be overengineered.

4/ Break the System into High-Level Components

Once the scope is clear and scale is known — I move to architecture.

I draw a basic skeleton:

- Clients → Web, mobile

- API Gateway → Handles authentication, rate-limiting

- Load Balancer → Distributes incoming traffic

- Microservices → Auth, User, Post, Feed, Media, Notifications

- Databases → SQL, NoSQL, blob storage

- Cache Layer → Redis or Memcached

- Asynchronous Processing → Queues + Workers

Simple is scalable.

I avoid buzzword-heavy diagrams and focus on showing how components talk to each other.

5/ Pick the Right Database Based on Use Case

I ask myself:

- Do I need ACID transactions or eventual consistency?

- Is my workload read-heavy or write-heavy?

- Do I need flexible schema or fixed schema?

- Can I survive with eventual consistency?

📌 Example:

- Banking system → SQL (you must have consistency)

- News feed → NoSQL (fast reads > strict consistency)

I justify every tech choice. I don’t say “MongoDB is good”   
  
 I say, “I chose MongoDB because my write-heavy workload benefits from flexible schema and fast inserts.”

6/ Scale the System for Growth

Here I talk about how I’d handle millions of users.

- Load balancing across multiple app servers

- CDNs for static files (videos, images, CSS, JS)

- Caching to reduce database load (Redis for sessions, Memcached for feed)

- Sharding the DB when vertical scaling hits its limit

- Database replication for read-scaling

- Asynchronous queues for background jobs (emails, notifications, video encoding)

Example:

“Video uploads are slow, so I’ll queue them and notify the user once encoding is complete.”

This shows I know where the bottlenecks are and how to handle them gracefully.

7/ Design for Failures (Because Real Systems Break)

I always address this because no real-world system is perfect.

- What if Redis goes down? → Fallback to DB with degraded performance.

- What if one microservice crashes? → Circuit breaker + retry mechanism.

- What if traffic spikes 10x? → Auto-scaling + rate limiting.

Great engineers build for failure.

8/ Monitoring, Logging, and Observability

Most people skip this. I don’t.

- I talk about logs (structured, searchable, scoped to trace IDs)

- I talk about metrics (latency, error rate, traffic, CPU, DB throughput)

- I talk about alerts (on-call setup, PagerDuty thresholds)

Why?

Because once you ship, you have to keep it running.

9/ Answer Like an Engineer

When I answer, I:

- Speak out loud to show my thought process

- Ask questions to clarify the scope

- Justify trade-offs with real-world context

- Keep track of time, you get ~45-60 mins to deliver a complete design

I’m not here to flex buzzwords.

I’m here to show that I can solve real business problems with clean, scalable, and maintainable systems.

TL;DR:

✅ Don’t memorize diagrams

✅ Think in terms of trade-offs

✅ Focus on clarity, not complexity

✅ Keep your answers grounded in why, not what

✅ Practice explaining things like you would to your teammates

The best system design answers are structured, clear, and confident