

How I'd Prepare for a System Design Interview if I Were Starting From Scratch



HELLO INTERVIEW

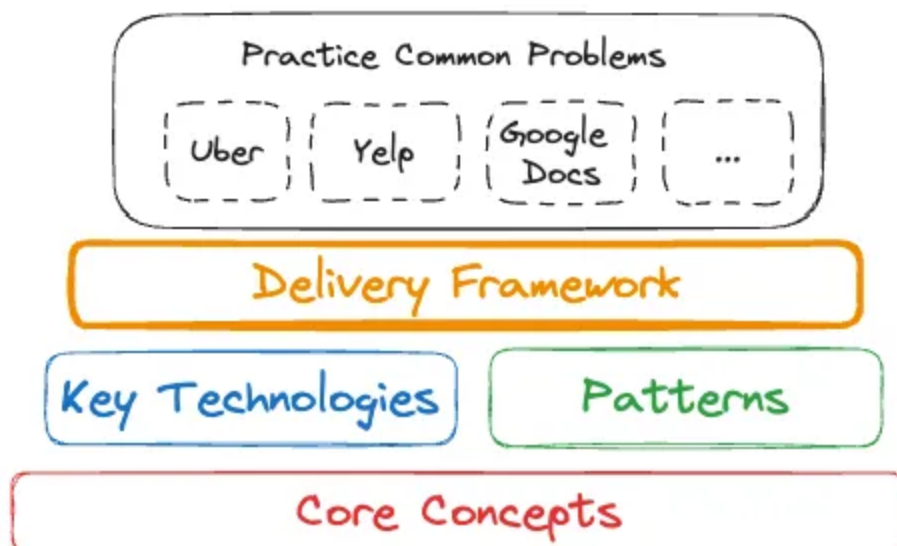
APR 10, 2025

♡ 94

💬 3

🔄 7

Sha



You just got an email from your dream company—an interview scheduled for a month from today. While you know how to prepare for coding interviews (hello, LeetCode) the looming system design interview has you nervous.

Maybe you're a mid-level engineer facing your first system design challenge, or perhaps you're a senior who's been out of the interview circuit for a while. You've worked at big companies, but only ever focused on your narrow part of the system. 'more you read online, the more overwhelming it seems.

Take a deep breath, you're not alone. As someone who has conducted over 500 system design interviews, I can tell you that this anxiety can melt away once you have a clear path to follow.

Thanks for reading Hello Interview!
Subscribe for bi-weekly content on SWE
interviews & hiring

Subscribe

The Four-S


- 1. Understand what
- 2. Refresh the fund
- 3. Master the build
- 4. Work backwards

Let's break down each

What is th

A system design interview is a type of interview that solve real-world problems. It tests your skills, system design and decision-making process (there often isn't one!). They want to see how you frame ambiguous problems, make reasonable trade-offs, and think about scalability and performance.

There are actually different flavors of system design interviews. The most common is Product Design (think "design Uber's backend"), but you might also encounter Infrastructure Design (like designing a rate limiter), Object-Oriented Design (class



Discover more from Hello Interview

Fresh insights on SWE interview landscape, in your inbox every 2 weeks.

Over 3,000 subscribers

Subscribe

By subscribing, I agree to Substack's [Terms of Use](#), and acknowledge its [Information Collection Notice](#) and [Privacy Policy](#).

Already have an account? [Sign in](#)

w?
able system
algorithm
s, and
fect answer

structures for a parking lot system), or Frontend Design (architecting a spreadsheet app). This prep strategy is effective for the first two, which make up the vast majority of interviews you'll face. The key to passing is delivering a working system—not getting lost in unnecessary details or failing to address the core requirements.

While every company is slightly different, they're all evaluating you against permutations of the same 4 axes:

1. **Problem Solving:** How well you identify, understand, and prioritize the core challenges in a system.
2. **Solution Design:** Your ability to create scalable architectures while balancing performance, maintainability, and cost trade-offs.
3. **Technical Excellence:** Your depth of technical knowledge and mastery of relevant tools and best practices.
4. **Communication:** How clearly you explain complex technical concepts and design decisions to various stakeholders.

Lastly, I'd strongly recommend you follow this [Delivery Framework](#) so that you stay focused and on track in the interview.

Refresh the Fundamentals

Before diving into complex system design, you need to master certain fundamentals. Most online guides will overwhelm you with endless lists of rarely used theorems or algorithms, but here's what actually matters to build a solid foundation:

1. Storage Fundamentals

You should be able to discuss various data storage models and their appropriate use cases. Be prepared to explain how data is organized in relational databases (normalized tables with relationships) versus document stores (nested JSON-like structures), and

key-value stores (simple lookup mechanisms). Your interviewer will expect you to understand ACID properties for transactional systems versus BASE principles for distributed databases, and to articulate when each storage paradigm makes sense based on access patterns and consistency requirements.

2. Scalability Fundamentals

You should be comfortable explaining both vertical scaling (beefier machines) and horizontal scaling (more machines) approaches. Be ready to discuss read/write segregation strategies and how they optimize different workloads. Your knowledge should include partitioning/sharding techniques for distributing data effectively across nodes. When the "celebrity problem" (handling hotspots) comes up, you'll want to articulate solutions clearly. Load balancing algorithms and consistent hashing techniques should be in your toolkit for distributing traffic and data efficiently.

3. Networking Essentials

You should be conversant in HTTP/HTTPS protocols, TCP/UDP differences, DNS resolution, and API design principles. Expect to discuss the request-response lifecycle and common network topologies. Your interviewer will appreciate it when you can clearly differentiate between REST, GraphQL, gRPC, and understand the role Websockets and SSE play in real-time communication systems.

4. Latency, Throughput, and Performance

You should be able to recall approximate latency numbers for common operations (memory access, disk reads, network calls). When discussing system requirements, demonstrate your ability to understand throughput limitations and perform basic capacity planning calculations. Be prepared to identify potential performance bottlenecks and propose solutions across a distributed architecture, ie. caching, sharding writes, etc.

5. Fault Tolerance & Redundancy

You should recognize and communicate that failures are inevitable in distributed systems. Be ready to discuss replication strategies and failure detection mechanisms. Your design should incorporate redundancy at appropriate levels (server, rack, datacenter) and be able to recover from failures gracefully.

6. CAP Theorem

You should be able to explain the fundamental trade-offs between Consistency, Availability, and Partition tolerance in distributed systems. When discussing database technologies, demonstrate your ability to classify them according to CAP characteristics and articulate which properties to prioritize based on specific business requirements and use cases. To make this simpler, given partition tolerance is a must, you just need to be able to decide whether your system needs strong consistency (like ticketbooking or banking apps) or whether eventual consistency is fine (everything else).

Understand the Building Blocks

At its core, system design is just combining components, like legos. You need to know which building blocks are available in your toolkit. Here are the main ones worth knowing.

1. Server

The computational backbone of your system. Servers process requests, execute business logic, and communicate with other components. Understand the difference between monolithic, microservice, and serverless architectures but don't over think this, this is just your compute!

2. Database

The persistent storage layer. Understand different data storage paradigms and their trade-offs. Relational databases (PostgreSQL, MySQL) excel at maintaining data integrity and complex relationships through structured schemas, transactions, and joins. Document databases (MongoDB), wide-column stores (Cassandra), and key-value stores (DynamoDB, Redis) offer different organization models for different access patterns. Modern databases across paradigms increasingly support both ACID transactions and horizontal scaling, so focus on understanding data modeling

approaches, indexing strategies, and query optimization techniques that match your application's read/write patterns and consistency requirements.

3. Cache

The speed layer. Implement caching (Redis, Memcached) to reduce database load and improve response times. Understand cache invalidation strategies, TTL policies, and the trade-offs between cache hit rates and data freshness. Know common patterns like cache-aside, read-through, and write-through.

4. Message Queue

The asynchronous communication layer. Use message queues (Kafka, RabbitMQ, SQS) to decouple services, handle traffic spikes, and implement reliable background processing. Understand at-least-once vs. exactly-once delivery semantics and how to handle failures.

5. Load Balancer

The traffic distribution layer. Deploy load balancers to distribute requests across multiple servers, improving availability and scalability. Know common algorithms (round-robin, least connections, consistent hashing) and how they affect system behavior.

6. Blob Storage

The unstructured data layer. Use object storage (S3, Google Cloud Storage) for files, images, videos, and other binary data. Understand access patterns, lifecycle policies, and cost optimization strategies for large-scale storage.

7. CDN

The edge delivery layer. Implement Content Delivery Networks to cache static assets closer to users, reducing latency and backend load. Know how to configure cache headers, invalidation strategies, and edge computing capabilities for global scale.

Work Backwards from Common Problem

With the fundamentals in place, you should work bottom-up from common problems. The key is to not passively consume content though. Watching videos and reading breakdowns will only get you so far.

Each time you come across a concept you don't know, dig deeper, fan out and ask ChatGPT, Google, etc. to close the gap. Once you've done the first 3-4 problems, then start to try them on your own before jumping to reading a solution.

This tests your ability to apply theoretical knowledge to practical scenarios, identify bottlenecks, and make reasonable trade-offs under constraints. It also reveals gaps in your understanding that you might not discover through passive learning.

The 10 problems we recommend, in order, are:

1. [Design a URL Shortener \(Bitly\)](#) - Tests your understanding of hashing, databases, and caching.
2. [Design Dropbox](#) - Tests file storage, synchronization, and metadata management.
3. [Design Ticketmaster](#) - Tests concurrency, race conditions, and transactional integrity.
4. [Design a News Feed](#) - Tests content delivery, personalization, and real-time updates.
5. [Design WhatsApp](#) - Tests real-time communication, presence detection, and message delivery.
6. [Design LeetCode](#) - Tests code execution environments, scaling compute, and security.

7. [Design Uber](#) - Tests geospatial indexing, matching algorithms, and real-time updates.
8. [Design a Web Crawler](#) - Tests distributed systems, scheduling, and politeness policies.
9. [Design an Ad Click Aggregator](#) - Tests high-throughput event processing and analytics.
10. [Design Facebook's Post Search](#) - Tests indexing, ranking, and search optimization.

For [Hello Interview Premium](#) users, I highly recommend practicing each of these problems yourself with our interactive [Guided Practice](#). If you don't have Premium, sweat, open up an Excalidraw whiteboard and start a timer, working through each problem on your own. While you won't get the benefit of instant feedback, you can read the breakdowns or watch the videos afterward to see where you might have made missteps.

As you work through these problems, you'll start to notice patterns emerge. By the time you've solved 5-6 design challenges, things will begin to feel more intuitive, and you'll quickly recognize which components and approaches fit specific scenarios. System design interviews, much like data structures and algorithms questions, ultimately test your ability to recognize problem patterns and apply appropriate solutions. Each new challenge is rarely completely novel, it's typically a recombination of familiar elements with specific constraints and trade-offs that you'll become increasingly comfortable navigating. It just takes repetition.

Lastly,, consider scheduling mock interviews to test your knowledge in a realistic setting. [Feedback from candidates](#) consistently shows that practice interviews are often the catalyst that makes theoretical knowledge click in practical application. The opportunity to receive real-time feedback from experienced interviewers can reveal blind spots in your preparation that might otherwise go unnoticed until your actual interview.

Most importantly, good luck! It's a grind, but you've got this!

Changelog

People are constantly asking us what's new with Hello Interview, so we're going to keep a changelog here to keep you up-to-date. Since our last update:

New Content

- [Flink deep dive \(Premium\)](#)
- [Design an Online Auction video \(Premium\)](#)
- [Design Instagram video \(Premium\)](#)

We've got more coming down the pipe that we're excited to share in our next update.

You can vote for what content you want to see next [here](#).

Subscribe to Hello Interview

By Hello Interview · Launched 4 months ago

Fresh insights on SWE interview landscape, in your inbox every 2 weeks.

By subscribing, I agree to Substack's [Terms of Use](#), and acknowledge its [Information Collection Notice](#) and [Privacy Policy](#).



94 Likes • 7 Restacks

Discussion about this post

Comments Restacks



Write a comment...



Lugman B Lukman Apr 16

Hi Stefan / Evan,

Is there anything you would add to this guide when interviewing for an AI Engineer r

♡ LIKE 💬 REPLY



Harshit Shah Apr 10

Hi Stefan / Evan,

Great job on the newsletter so far—really enjoying the updates! As an avid user of the platform, I always look forward to the changelog at the end of each post. I was wondering if it might be helpful to have a dedicated changelog section on the website as well—somewhere central where folks can easily access it anytime?

♡ LIKE 💬 REPLY



1 reply by Hello Interview

1 more comment...