kaustubh yerkade
Posted on Oct 3, 2024 • Edited on Oct 11, 2024

❤️ 23 🦄 1

System Design Fundamentals: A Complete Guide for Beginners

#systemdesign #architecture #kubernetes #docker

System design is easy !! The key is knowing what to use and when to use it. Once you get familiar with the tools and patterns, it becomes second nature. The complexity comes from balancing the specific needs of the system, but with experience, even that becomes intuitive! and In the end its nothing but ways to move and store data around,

Before you start designing anything, ask: **What problem are you solving?**

Is your system meant to handle millions of user transactions, or are you just trying to host a small website for friends? or a small e-commerce website your Aunt asked for selling her homemade cookies online, Or a large scale IPL streaming platform with millions of active users streaming content at the same time, This helps set the stage for everything else because designing for a global social network is way different from a personal blog.

Lets understand in simple terms-

1. **Purpose**
   First, what’s the system's vibe? Like, why does it even exist? Is it to handle big data, serve a million users, or just vibe out and handle basic stuff?

2. Squad Goals (**Users**)
   Who's pulling up? You got users or clients who need your system. Is it low-key just a few, or is it like a whole squad rolling deep? or millions ? This’ll help decide how extra your setup needs to be.

3. **Scalability**
   Okay, now think: can this system handle glow-up moments? If it gets real hype, can it flex and still perform, or will it just crash?

4. **Frontend**
   The drip is all about what users see. Clean UI is a must. Nobody wants to use a system that’s stuck in 2010. Keep it smooth, modern, and don’t sleep on mobile.

5. Brains (**Backend**)
   This is where the system's smart moves are made. Are we talking basic REST APIs, or is it more low-key microservices? You need it quick, efficient, and not crashing under pressure.

6. Cloud or Nah (**Infrastructure**)
   Are you on the cloud like AWS, Azure, GCP or chilling on your own servers? Most vibes are on cloud ‘cause who’s trying to babysit servers 24/7? Plus, it's way easier to scale when your app goes viral.

7. **Database**
   You gotta store the data. SQL if it’s structured, but NoSQL when you’re keeping it casual or dealing with too much randomness. Make sure it’s fast or users will ghost.

8. Firewall (**Security**)
   Security is key. You don’t want your system getting hacked, right? Strong passwords, encryption, and keep an eye on DDoS attacks like you would on toxic behavior.

9. Speed Check (**Performance**)
   Don’t have your users waiting around like bookmyshow did while coldplay. Optimize for speed. No one has time for buffering. Caching, load balancing, and CDNs are your BFFs here.

10. Backup Plan (**Resilience**)
    Always have a backup, for real. If something crashes, your users should never know. Redundancy, failover systems, and regular backups should be on point.

11. Tea Time (**Monitoring**)
    Stay woke on what's happening. Monitoring tools (like Prometheus, Grafana) are essential to keep your system from wildin'. You'll know if it's down before anyone else.

12. Glow Up (**Continuous Improvement**)
    You gotta stay in beta forever—constantly pushing updates, fixing bugs, and improving. Don't get too comfortable, or you'll get left behind.

When you break system design down into smaller, digestible steps, it's not as overwhelming as it might seem at first. Here's why it can feel easy:

- **Frameworks & Tools:** There are tons of powerful tools and frameworks available (like cloud platforms, monitoring tools, databases, and design patterns) that simplify a lot of the hard work.
- **Reusing Components:** Most systems follow similar patterns—authentication, databases, APIs, etc. Once you've done it once, the next time is easier.
- **Community Knowledge:** There's a wealth of knowledge online. Stack Overflow, GitHub, and blogs provide solutions to almost any problem you might face.
- **Scalability On Demand:** Cloud services like AWS, Azure, and Google Cloud make scaling as easy as flipping a switch. You don't have to manually manage servers anymore.
- **CI/CD Pipelines:** Automating testing and deployment makes pushing updates fast and error-free.

In today's tech-driven world, building robust, scalable, and efficient systems is a key.

Key Components of System Design

Let's break down the fundamental concepts of system design:

1. Scalability
Scalability is the ability of a system to handle increased load without compromising

performance. There are two types of scalability:

Vertical Scaling (Scaling Up): Increasing the capacity of a single machine (more CPU, memory, etc.).

Horizontal Scaling (Scaling Out): Adding more machines to distribute the load. This is often preferred for large-scale systems like social media platforms or cloud applications.

For example, Netflix uses horizontal scaling to distribute video streaming load across multiple servers worldwide.

2. Load Balancing
Load balancing helps distribute incoming traffic across multiple servers. This ensures no single server is overwhelmed and that users get fast, reliable service. Popular load balancers include:

Round-robin: Sends requests to servers in rotation.
Least connections: Sends requests to the server with the fewest active connections.
For instance, when a million users are accessing a website like Amazon, load balancers evenly distribute their requests to multiple servers.

3. Caching
Caching is about storing frequently used data in temporary storage (RAM) to speed up response times. By avoiding repetitive database queries, caching significantly improves system performance. Tools like Redis and Memcached are popular for this purpose.

Imagine you're building a Twitter-like feed system. When users repeatedly view the same posts, you can cache the results and serve them from memory instead of querying the database each time.

4. Database Design
Databases store the data your application needs. There are two primary types of databases:

SQL (Relational databases): Structured, with predefined schemas (e.g., MySQL, PostgreSQL). Ideal for transactional systems like banking.
NoSQL (Non-relational databases): Flexible and scalable (e.g., MongoDB, Cassandra). Ideal for unstructured or semi-structured data, like social media posts.

For instance, an e-commerce site might use SQL to manage orders and customers, while a NoSQL database could handle product reviews and recommendations.

5. Consistency, Availability, and Partition Tolerance (CAP Theorem)
The CAP theorem is a fundamental concept in distributed systems, and it states that a system can only guarantee two of the following:

Consistency: Every read gets the most recent write.
Availability: Every request receives a response, even if it's not the most recent data.
Partition Tolerance: The system continues to operate despite network partitions.

You need to decide what your system prioritizes. For example, a financial system would prioritize consistency, whereas a social media platform might favor availability.

6. Message Queues
Message queues are used to handle **asynchronous** tasks, allowing a system to process high volumes of requests smoothly. **Kafka and RabbitMQ** are common tools used for queuing messages in a system.

For example, if you’re building an online order system, once an order is placed, it’s sent to a message queue. This way, order processing happens in the background, and the user doesn’t have to wait for it to complete.

7. CDN (Content Delivery Network)
A CDN helps deliver content (like images, videos, or web pages) to users from servers that are geographically closer to them, reducing latency. Services like Cloudflare or AWS CloudFront are commonly used for CDN.

For instance, if someone in India is accessing a website hosted in the U.S., a CDN will serve cached content from a nearby server in India, improving load times.

8. Redundancy and Fault Tolerance
Redundancy involves having multiple instances of critical system components. If one instance fails, another can take over, ensuring fault tolerance. This is key to building systems that are resilient to failure.

For example, in a payment processing system, if one server goes down, the redundant system should kick in immediately so payments can continue being processed.

Step-by-Step System Design Process
Now that we’ve covered the basics, let’s look at a step-by-step process for designing a system:

Understand the Requirements:

Identify functional requirements (what the system should do) and non-functional requirements (scalability, reliability, etc.).

Example: If you’re building an online store, functional requirements could include user login, product listings, and checkout, while non-functional requirements might include handling 1,000 users at once.
Define the High-Level Architecture:

Start by sketching a high-level architecture of your system.
Example: For an e-commerce site, you might have:

- A frontend (web app or mobile app).
- A backend (handles API requests).
- A database (stores user, product, and order information).
- Load balancers to manage traffic.

Break Down Components:

Break the system into smaller components like-
user authentication,
product catalog,
payment gateway, etc.

Decide which components are services (microservices) or part of a monolith.

Choose Databases and Storage:

Choose SQL or NoSQL based on the type of data.

Example: A relational database for transactional data (orders) and NoSQL for dynamic data (product reviews).

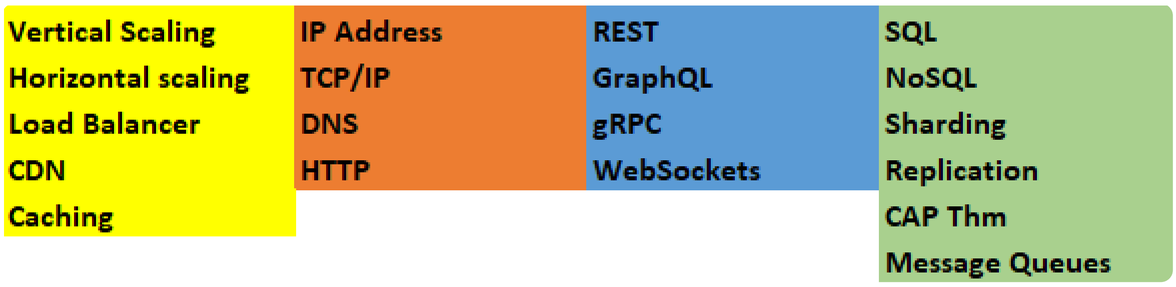
Plan for Scalability and Redundancy:

Implement load balancing, caching, and redundancy to ensure scalability and reliability.

Example: Use Redis for caching and implement a message queue to handle order processing in the background.

Monitor and Improve:

Use tools like Prometheus or Datadog to monitor your system’s performance and respond to any issues before users notice them.



Vertical Scaling	IP Address	REST	SQL
Horizontal scaling	TCP/IP	GraphQL	NoSQL
Load Balancer	DNS	gRPC	Sharding
CDN	HTTP	WebSockets	Replication
Caching			CAP Thm
			Message Queues

System Design Examples-

Here are some common system design examples that are widely used in interviews and real-world applications. Each example will give you an idea of how to design systems with specific challenges like scalability, availability, and fault tolerance:

1. Design a URL Shortener (e.g., bit.ly)

A URL shortener takes a long URL and generates a shorter version that redirects to the original URL. It's a simple but classic system design problem.

Key Considerations:
Database: Store mappings between original URLs and shortened versions.
Unique ID Generation: You need a way to generate unique short URLs (base62 encoding, hashing, etc.).
Caching: Use caching (e.g., Redis) for frequently accessed URLs to reduce database lookups.
Scalability: Handle millions of requests per second, especially when URLs go viral.
Expiration: Allow URLs to expire after a certain period.

Example High-Level Architecture:

- A frontend that accepts the long URL.
- A backend that generates the short URL and stores it in the database.
- Caching and load balancing for efficiency.

2. Design an Online Bookstore (e.g., Amazon)
An online bookstore needs to manage user accounts, display products (books), process orders, and handle payments.

Key Considerations:
Product Catalog: Store book details like title, author, price, etc., in a database.
Search Engine: Implement search functionality using a search engine like Elasticsearch.
User Authentication: Secure user login and registration.
Shopping Cart: Design a shopping cart that saves user selections.
Order Management: Handle order placement, payments, and notifications.
Recommendations: Implement a recommendation engine using collaborative filtering or machine learning.

High-Level Architecture:

- Microservices for user management, product catalog, shopping cart, and order processing.
- Database for product details (SQL) and customer interactions (NoSQL).
- Caching for frequently viewed products.

3. Design a Social Media Feed (e.g., Twitter, Facebook)
A social media feed displays posts (text, images, videos) from people a user follows, sorted by relevance or time.

Key Considerations:
Feed Generation: Design a service to generate personalized feeds in real-time.
Database: Store user posts and their relationships (followers/following).
Caching: Cache frequently accessed posts or popular content.
Scalability: Handle millions of users and real-time updates.
Sharding: Shard data across multiple databases to handle massive user bases.
Push vs Pull Model: Choose between pushing updates to users' feeds (more real-time but resource-heavy) or pulling when users check their feed.

High-Level Architecture:

- A backend service that handles post creation and feed generation.
- A real-time database (like Cassandra) for storing posts and user connections.
- Caching system (e.g., Redis) for popular posts.

4. Design a Ride-Sharing System (e.g., Uber, Lyft)

A ride-sharing platform connects drivers and passengers, showing available rides, estimating fares, and ensuring smooth transactions.

Key Considerations:
Real-Time Location: Track drivers' and passengers' locations in real-time.
Matching Algorithm: Match riders to the nearest available driver.
Database: Store ride details, user profiles, trip history, and driver availability.
Scalability: Handle high traffic, especially in busy areas or times.
Surge Pricing: Implement dynamic pricing based on demand.
Fault Tolerance: Ensure that the system works even if some services fail.

High-Level Architecture:

- A location-based service that updates drivers' locations in real-time.
- A matching service that connects riders and drivers.
- Database for trip details, driver profiles, and payments.
- Real-time communication system (e.g., WebSockets) for trip updates.

5. Design a Video Streaming Platform (e.g., YouTube, Netflix)

A video streaming service allows users to upload, view, and stream video content with high availability and low latency.

Key Considerations:
Video Storage: Efficiently store large volumes of video data.
Video Encoding: Transcode videos into multiple formats to support different devices.
CDN (Content Delivery Network): Distribute video content across geographically distributed servers to reduce latency.
Recommendation Engine: Suggest videos based on user preferences and watch history.
Load Balancing: Ensure requests are distributed across servers to avoid overload.
Scalability: Handle millions of concurrent users watching content.

High-Level Architecture:

- A backend for handling video uploads, encoding, and metadata storage.
- A CDN to deliver video content globally.
- A recommendation service powered by machine learning algorithms.
- A caching layer to store metadata for faster video retrieval.

6. Design a File Storage System (e.g., Google Drive, Dropbox)

A file storage system allows users to upload, download, and share files with others.

Key Considerations:
Storage: Handle large files and store them efficiently.
File Metadata: Store metadata like file size, type, and ownership.
Version Control: Allow users to keep different versions of their files.
Access Control: Set permissions for shared files and folders.
Redundancy: Use multiple servers or data centers to ensure files aren't lost.
Fault Tolerance: Ensure files are still accessible in case of hardware failures.

High-Level Architecture:

- A file upload service that stores files in a distributed file system (e.g., Amazon S3).
- Metadata service to manage file information.
- Caching layer for frequently accessed files.
- Redundant storage to avoid data loss.

7. Design a Chat Application (e.g., WhatsApp, Slack)

A chat application allows real-time messaging between users, with support for text, media, and notifications.

Key Considerations:
Real-Time Messaging: Implement real-time communication using technologies like WebSockets.
Database: Store chat histories, user details, and media files.
Message Queues: Handle asynchronous message delivery using message queues like Kafka or RabbitMQ.
Group Chats: Support group messaging and notification systems.
Scalability: Handle millions of users sending messages simultaneously.
End-to-End Encryption: Ensure that messages are secure and private.

High-Level Architecture:

- A messaging service powered by WebSockets for real-time delivery.
- A database (e.g., Cassandra) for storing chat history.
- Caching for frequently accessed chat histories or user info.
- A notification service for real-time alerts on new messages.

8. Design a Search Engine (e.g., Google Search)

A search engine indexes web pages and provides relevant search results in response to user queries.

Key Considerations:
Web Crawlers: Design bots that crawl the internet and index content.
Indexing: Store the indexed data efficiently to ensure fast searches.
Ranking Algorithm: Implement algorithms like PageRank to rank pages by relevance.
Caching: Cache frequently searched queries to reduce database load.
Scalability: Handle billions of searches per day.
Latency: Ensure low latency for user queries by distributing data across multiple servers globally.

High-Level Architecture:

- Web crawlers that index content.
- A search index stored in a highly optimized database (e.g., Elasticsearch).
- Ranking and recommendation engine to improve search results.
- Distributed architecture with caching to reduce query times.

9. Design a Payment Gateway (e.g., Stripe, PayPal)
A payment gateway processes financial transactions securely between customers and merchants.

Key Considerations:
Payment Processing: Handle credit/debit cards, digital wallets, and bank transfers.
Security: Implement PCI compliance, encryption, and tokenization to protect sensitive data.
Transaction Logging: Maintain logs of all transactions for auditing and troubleshooting.
Fraud Detection: Use algorithms to detect and prevent fraudulent transactions.
Scalability: Handle a large volume of transactions simultaneously without delays.
Redundancy: Ensure high availability and zero downtime, especially during high traffic events like Black Friday.

High-Level Architecture:

- A payment service that interacts with banks and card networks.
- A secure database for storing transaction details.
- Fraud detection service using machine learning.
- Load balancing to distribute transactions across multiple servers.

Each of these examples requires a thoughtful approach to **scalability, fault tolerance, and real-time performance** . By understanding the challenges of each system, you can design solutions that are resilient, scalable, and efficient.

Looking to level up your system design skills? Stay tuned & click follow for more in-depth posts on advanced system design topics like microservices architecture, database sharding, and real-world case studies! leave your valuable comments down below & let me know your thoughts

Top comments (3)

Jessilyneh • Jan 4

thank you for this amazing content 🫶

BADRI NARAYANAN • Dec 25 '24

Great article for system design

Chandan Maurya • Oct 3 '24

Well explained!

Code of Conduct • Report abuse

kaustubh yerkade

Hi 👋 I am kaustubh Welcome to my blog.... I'm a DevOps engineer with a passion for automation, agile tech , continues improvement & scalability.

JOINED
May 29, 2020

More from kaustubh yerkade

Terminus : Streamline Your Server Access and Management

#linux #devops #kubernetes #cli

Mastering RHEL Linux: A Comprehensive Cheat Sheet for Beginners & Experts

#linux #devops #kubernetes #docker