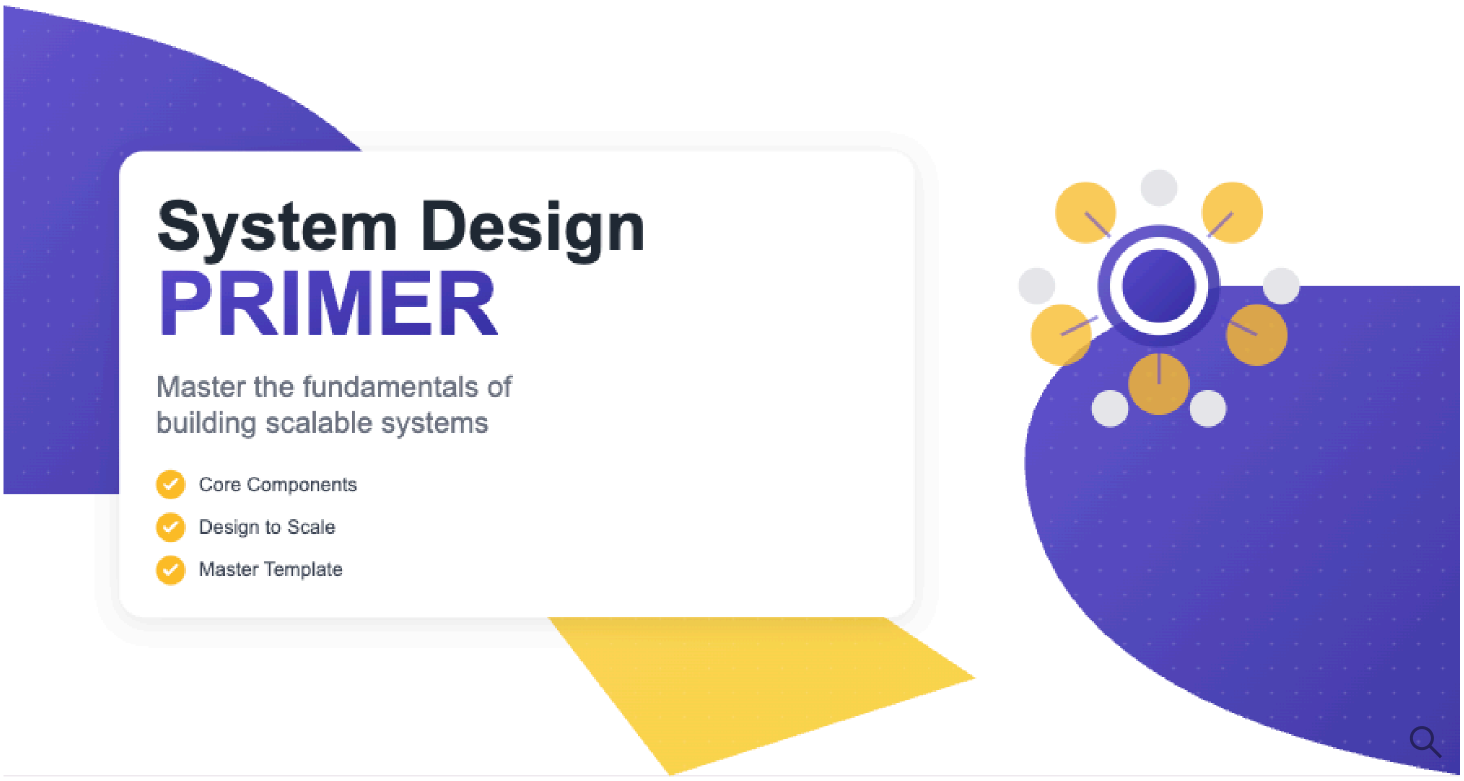
System Design Primer



Introduction

Welcome to the System Design Primer by our team here at System Design School.

System design knowledge is essential for all developers to learn for two main reasons:

1. System design interviews are asked when interviewing as most companies, especially in big tech
2. System design is what separates good from great engineers - anyone can code, but not anyone can design robust, scalable systems

That being said, "system design" is a relatively catch-all term... so let's explore more about what system design actually means.

What Is System Design?

Have you ever wondered how big tech companies like Google, Amazon, and Netflix build systems that effortlessly handle billions of users, terabytes of data, and massive spikes in traffic - all while staying fast, reliable, and secure? This is all done through the development of robust systems that work to be as efficient as possible, at as large a scale as possible. And the process of building these systems is what we refer to as system design.

System design is the blueprint behind every successful application or platform you've ever used. It involves thinking about how to put all the moving parts of a system together - databases, APIs, caching layers, load balancers, distributed queues, and more - so they work seamlessly to deliver a smooth user experience.

But system design is more than just plugging in technical components. It’s about making critical decisions that directly impact how a system performs, scales, and adapts over time. How do you ensure your system stays fast when traffic doubles overnight? How do you make it fault-tolerant so users aren’t affected if a server crashes? How do you store and retrieve data quickly while balancing costs? These are the kinds of questions system design addresses.

At its core, system design is about solving problems at scale. It’s not just about getting a feature to work; it’s about ensuring it works for millions of users in the most efficient and reliable way possible. This requires balancing competing priorities: speed versus cost, consistency versus availability, simplicity versus flexibility. The magic of system design lies in finding the right trade-offs for your specific use case.

What Are System Design Interviews?

As you progress in your career, writing code no longer becomes your main job - after all, with AI advancements, writing code is becoming easier than ever. Instead, companies need engineers who can step back and look at the big picture. Can you design something that works seamlessly under high traffic? Can you make decisions that balance cost, speed, and reliability? This is what gets assessed in a system design interview.

System design interviews are especially important for mid-level and senior engineering roles, but are also becoming more common at the new graduate level (and we expect this trend to continue as AI grows). At these levels, you’re expected to contribute to the architecture of applications, lead technical discussions, and make design choices that impact the entire system. These interviews simulate those real-world scenarios, giving companies a glimpse into how you handle challenges like scalability, fault tolerance, and performance.

System design, as we’ve discussed, is a lifetime pursuit - each system itself requires thousands (or even more) hours of work. So how can we demonstrate that we have this knowledge in a 45 minute interview?

That’s the real challenge of the system design interview - knowing what to answer, and how to answer it. There's no real guide so most candidates have to fail multiple interviews before even knowing what to expect.

That’s where we come in - with this completely free primer, our courses, and our interactive interview practice, you can both learn the concepts and master the interview process, so you can land your dream job.

What Are System Design Interviews Testing?

A common misconception is that system design interviews are testing knowledge of system design content. At their core, system design interviews are really testing your soft skills, primarily:

- Problem-Solving Skills: Can you take a vague, open-ended problem and break it down into smaller, solvable parts?
- Big-Picture Thinking: Do you understand how different components of a system interact? Can you design systems that are scalable, reliable, and maintainable?

- Trade-Off Analysis: Every design decision has pros and cons. Can you explain why you chose one approach over another?
- Communication: It's not just what you design - it's how well you explain it. Clear communication is key to collaborating with teams and stakeholders in real-world settings.
- Adaptability: System design is rarely a straight path. Can you adjust your approach based on new constraints or feedback?

Where Do I Start?

Good news - you're in the right place.

This primer is organized in a logical way so that you get the hang of system design concepts, and then understand the framework in which you'll solve system design interviews.

If you're new to system design, check out the main components section.

If you're already pretty familiar with the main components and want to learn how to structurally solve any problem, check out our step-by-step interview walkthrough where we go through the entire interview process and how to tackle each step.

If you already know the basics and want a quick way to get the gist of solving system design problems, check out our

- Understanding Core Design Challenges
- How to Scale a System

If you have the interview tomorrow and want to learn ***the one thing that solves 90% of the problems*** you'll run into, check out our **System Design Master Template**.

Main Components

Many of the most common system design problems have been solved - in fact, some of them are so common that engineers have developed tools for these that fit into every system, which we refer to as **components**.

A system design interview isn't about solving problems from scratch - it's about knowing what components you need to piece together. And luckily for you, this section highlights the most common components that will be used in every system design problem. All you need to do is learn how these technologies work, and when to use them, and you're good to go!

Each components is broken down into sections as follows:

- Laying the Groundwork: Explains the kind of system you might be working with, what common problem you will run into, and how the component solves this problem - this is how you know when to use the component
- Technical Explanation: High-level walkthrough of how the components works from a technical perspective - your interviewer may ask you to explain the component you're using, especially if you're more junior, so it's good to have a technical understanding
- Common Implementations: Every component has a variety of designs - for most interviews, you'll have to pick one and justify why you chose it, so learn a few of these

Microservices

Laying the Groundwork

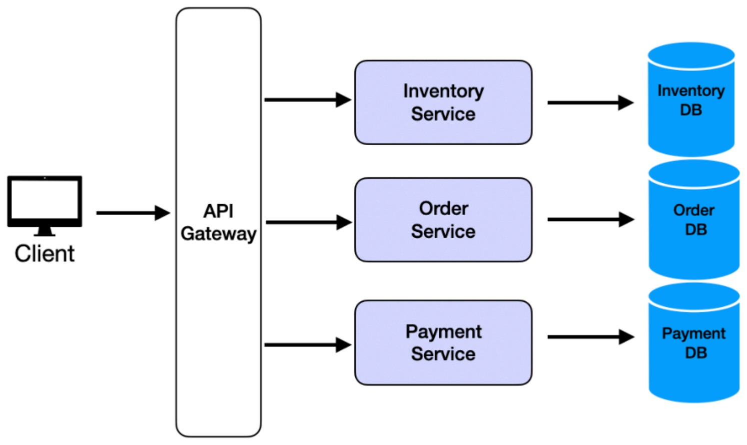
Services are where the application code and business logic lives. While you can structure your application as either a monolith or microservices, most system design interview problems deal with large-scale systems that benefit from a microservices architecture.

In a monolithic architecture, all of your application's functionality lives in a single codebase and runs as a single process. While this works well for small applications, it becomes problematic at scale because:

- Changes to any part require redeploying the entire application
- A bug in any component can bring down the whole system
- The entire codebase becomes harder to understand as it grows
- Different components can't be scaled independently

This is where **microservices** come in. Microservices break down your application into smaller, independent services that each handle a specific business function. For example, in an e-commerce system, you might have separate services for:

- Product catalog
- Shopping cart
- User authentication
- Order processing
- Payment processing



Technical Explanation

At a high level, microservices work by:

1. Service Independence: Each service runs as its own process and can be deployed independently
2. API Communication: Services communicate with each other through well-defined APIs, typically REST or gRPC
3. Database Isolation: Each service typically manages its own database, preventing direct database coupling

4. Independent Scaling: Services can be scaled independently based on their specific load requirements

In an interview, you should also be able to explain:

- Service Discovery: How services find and communicate with each other
- Data Consistency: How to maintain data consistency across services
- Fault Isolation: How failures in one service are contained and don't cascade to others

Common Implementations

Spring Boot

Spring Boot is a popular Java framework for building microservices.

Key Features:

- Built-in support for REST APIs and service discovery
- Extensive ecosystem of libraries and tools
- Ideal for Java-based microservices requiring robust enterprise features

Node.js with Express

Node.js with Express is a lightweight option for building microservices.

Key Features:

- Fast development and deployment
- Large ecosystem of npm packages
- Ideal for JavaScript/TypeScript microservices needing quick iteration

Go with Gin

Go with Gin is known for high performance and simplicity.

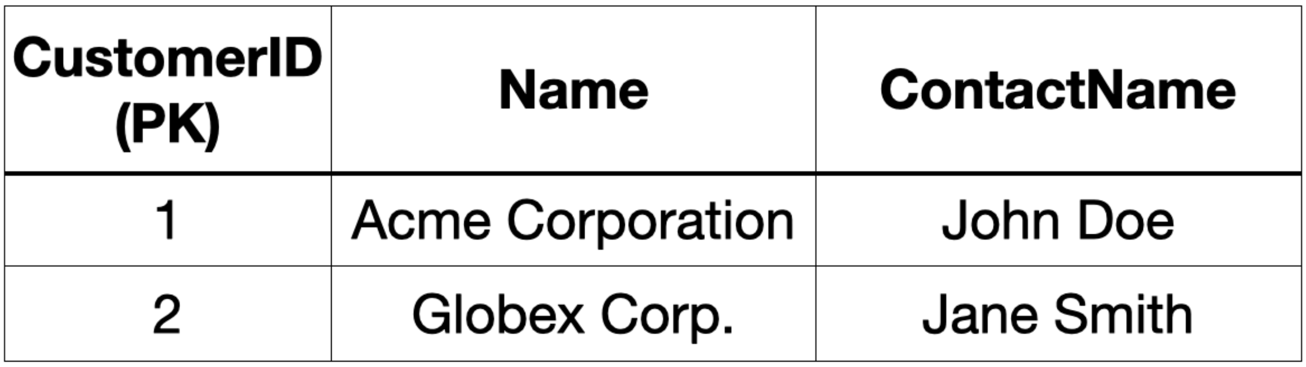
Key Features:

- Excellent performance characteristics
- Built-in concurrency support
- Ideal for microservices requiring high throughput and low latency

Relational Database

Laying the Groundwork

At the heart of most applications lies data, and in many cases, that data needs to be structured in a way that’s easy to understand, query, and maintain. Enter the **relational database**, a workhorse for handling structured data in countless systems.



CustomerID (PK)	Name	ContactName
1	Acme Corporation	John Doe
2	Globex Corp.	Jane Smith

A relational database organizes data into tables, which look a lot like spreadsheets. Each table contains rows (records) and columns (fields). This structure makes it straightforward to link different pieces of data together, thanks to something called relationships. For example, in an e-commerce system, you might have one table for customers, another for orders, and a relationship between them to keep track of who bought what.

But why would you choose a relational database in a system design interview (or real life)? It's all about consistency and structure. If your app deals with data that must stay accurate and interrelated, like financial transactions, user profiles, or inventory levels, a relational database is often the right choice. It ensures:

- Data Integrity: No weird duplicates or orphaned records thanks to constraints like primary and foreign keys
- Powerful Querying: SQL allows you to filter, join, and analyze data with ease
- ACID Properties: Transactions are guaranteed to be Atomic, Consistent, Isolated, and Durable - essential for systems where accuracy matters

If your interviewer gives you a problem where data needs to be consistent, relational databases are usually your best bet.

Technical Explanation

Relational databases are built around a simple but powerful idea: data is organized into tables, and relationships between those tables are defined through keys.

Here's a quick rundown of the main components in a relational database:

- Tables: Think of tables as the basic building blocks. Each table represents an entity, like Users or Orders. Columns define what kind of data is stored (e.g., name, email, order_date), and rows hold the actual data.
- Primary Key: A unique identifier for each row. For example, every user might have a unique user_id.
- Foreign Key: A reference to a primary key in another table, creating relationships. For instance, an order_id in an Orders table might reference a user_id in the Users table.

There are also a few kinds of relationships you should know about:

- One-to-One: Each record in Table A links to exactly one record in Table B (e.g., user profiles and user settings)

- One-to-Many: One record in Table A links to many records in Table B (e.g., a single user can place multiple orders)
- Many-to-Many: Many records in Table A link to many in Table B, often using a third table to manage the relationship (e.g., students and classes)

In a system design interview, you may be asked questions about relational databases such as the relationships between entities you define, as well as asked about the schema, which includes info about the tables and keys. For a more in-depth look, check out our course (course here)

Common Implementations

PostgreSQL
Postgres is an open-source relational database known for its robustness and advanced features, and is one of the industry standards for relational databases.

Key Features:

- Supports complex queries and custom extensions
- Excellent for analytical workloads alongside traditional transactional uses
- Great for projects where you need flexibility, reliability, and open-source licensing

MySQL
MySQL is another widely-used open-source database, and is also an industry standard.

Key Features:

- Fast and efficient for read-heavy workloads
- Well-supported by many hosting platforms
- Reliable and easy-to-deploy database

SQLite
SQLite is a lightweight, serverless relational database, but is typically not used for large-scale applications.

Key Features:

- Embedded directly into applications
- Zero setup, as it is just a single file
- Ideal for mobile apps, small projects, or prototypes where simplicity is key

NoSQL Database

Laying the Groundwork

Not all data is neat and structured like rows in a spreadsheet. Sometimes, your data is messy, dynamic, or just doesn’t fit nicely into the table structure that’s expected in a relational database. That’s where **NoSQL databases** come in. They’re designed to handle unstructured (or semi-structured) data and can scale horizontally to manage massive amounts of it.

Imagine building a social media platform. Users post photos, write comments, add hashtags, and even store metadata like geolocations and device types. Trying to cram all this varied data into rigid tables would be a nightmare. A NoSQL database gives you the flexibility to store and retrieve this data without forcing it into a predefined schema.

NoSQL databases are perfect when:

- Schema Flexibility: Your data changes frequently, or you don't know its structure ahead of time
- High Scalability: Your app needs to handle massive amounts of traffic and data
- No Complex Relationships: You don't need complex relationships between data

In a system design interview, you'll need to justify why a NoSQL solution makes sense over a relational database. NoSQL databases trade the strict structure and complex querying provided by relational databases in exchange for flexibility and scalability. Many of them are distributed by design, meaning they can handle massive datasets and high traffic by spreading the load across multiple servers. For a more comprehensive look, check out our databases course (course here)

Technical Explanation

While there is really only one type of SQL database, which we call relational, NoSQL has multiple categories of databases, based on how they store and organize data. Here's how the most important ones work:

Key-Value Stores

Key-value stores are the simplest form of NoSQL, where all data is stored as key-value pairs, like a giant dictionary.

For example, the key could be a user like `user_123` and the value could be `{"name": "John Doe", "email": "john@example.com"}` . This is an easy way to map the name and email attributes to the user, without needing a table structure.

Key-value stores are commonly used for caching, session management, or storing user preferences.

Document Databases

Document databases do what the name implies, storing data within documents, typically using JSON formats. Each document contains key-value pairs and can nest data, making it flexible and self-contained. It's basically a more robust version of key-value stores.

For example, a user profile might look like this:

```
{
  "user_id": "123",
  "name": "John Doe",
  "email": "john@example.com",
  "posts": [
    {"post_id": "1", "content": "Hello World!"},
    {"post_id": "2", "content": "I love SystemDesignSchool!"}
  ]
}
```

Document databases are commonly used for content management systems, user profiles, or any application with dynamic, hierarchical data.

Column-Family Stores

Column-family stores use rows and columns for data storage, but unlike in relational databases, columns are grouped into families. This design is optimized for querying large datasets.

For example, a table for user analytics might store one row per user but have hundreds of columns for different metrics. Common use cases include working with time-series data, logs, or analytics.

Graph Databases

Graph databases represent data as nodes (entities) and edges (relationships), which might seem familiar if you've prepared for coding interviews. This type of NoSQL database is ideal for modeling highly interconnected data.

For example, a graph database is optimally used for a social network that might store users as nodes and friendships as edges. Aside from social networks, common uses of graph databases include recommendation systems, and fraud detection.

Common Implementations

MongoDB (Document Store)

MongoDB is a popular document-based NoSQL database.

Key Features:

- Flexible schema: Add or remove fields without downtime
- Rich querying capabilities: Filter, sort, and aggregate data easily
- Ideal for apps with dynamic data structures, like e-commerce and social media platforms

Redis (Key-Value Store)

Redis is an in-memory key-value store designed for speed, and as mentioned earlier in our cache section, is also a common in-memory cache implementation.

Key Features:

- Blazing-Fast Performance: Processes data with microsecond latency for real-time applications
- Advanced Data Structures: Supports lists, sets, sorted sets, and more for versatile use cases
- Ideal for caching, session storage, and real-time leaderboards

Apache Cassandra (Column-Family Store)

Apache Cassandra is a distributed database optimized for high availability and scalability.

Key Features:

- High Availability: No single point of failure ensures consistent uptime
- Massive Dataset Handling: Efficiently manages and queries large-scale data across distributed systems
- Ideal for high write-throughput applications like logs, or large-scale analytics systems.

Amazon DynamoDB (Key-Value/Document Store)

Amazon DynamoDB is a fully managed, serverless NoSQL database by AWS.

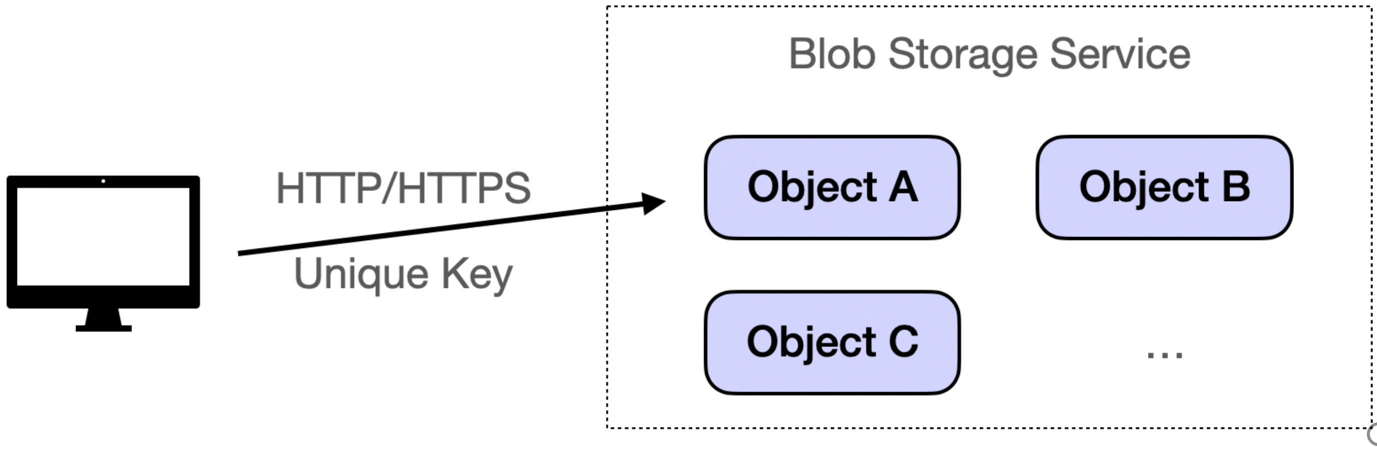
Key Features:

- Auto-Scaling: Seamlessly adjusts throughput to match traffic demand, ensuring cost-efficiency
- Low-Latency Performance: Provides near-instant responses, even under heavy workloads
- Ideal for serverless architectures or apps with unpredictable traffic patterns, like e-commerce platforms that have spiked traffic during sales

Object Storage

Laying the Groundwork

As applications and systems grow, so does the need to store large amounts of unstructured data - think images, videos, backups, and logs. Traditional storage solutions like file systems or databases often struggle to keep up with the scalability and flexibility needed for such massive datasets. That's where **object storage** comes in.



Object storage organizes data as objects, rather than files or blocks. Each object includes:

- Data: The file or blob itself, such as an image or video
- Metadata: Descriptive information about the object, like its size, content type, or access permissions
- Unique Identifier: A globally unique key to locate and retrieve the object

This design makes object storage highly scalable, cost-effective, and ideal for handling unstructured data. Unlike hierarchical file systems, object storage is flat, with no directories or folders - it's all about storing and retrieving objects using their unique keys.

You might choose object storage in scenarios like:

- Static Asset Hosting: Websites or apps serving images, videos, or documents
- Backups and Archives: Long-term storage of data that doesn't need frequent updates
- Big Data and Analytics: Storing large datasets for analysis or machine learning

Technical Explanation

At a high level, here's how object storage works:

- Data as Objects: Each file is stored as an object, which includes the file itself (binary data), metadata describing it, and a unique key

- Flat Storage Architecture: There are no folders or hierarchies. Instead, objects are stored in "buckets" or "containers" and are accessed using unique keys
- RESTful APIs: Object storage systems are typically accessed using APIs for storing, retrieving, or deleting objects
- Scalability: Object storage scales horizontally by adding more servers or nodes, distributing objects across them automatically, making it possible to handle massive amounts of data
- Durability and Redundancy: Data is often replicated across multiple servers or regions, ensuring high durability even if some nodes fail

Object storage sacrifices low-latency operations (like in databases or file systems) for scalability and cost-efficiency, making it perfect for storing data that doesn't require frequent updates.

In an interview, you might be asked to explain additional concepts:

- Versioning: Keeping multiple versions of the same object to protect against accidental deletions or overwrites
- Lifecycle Policies: Automatically transitioning objects to cheaper storage tiers or deleting them based on age or access patterns
- Consistency Models: Understanding eventual consistency vs. strong consistency in object storage systems

Common Implementations

Amazon S3 (Simple Storage Service)
S3 is a scalable, secure, and durable object storage service from AWS, and is the most used object storage in the industry.

Google Cloud Storage
Google Cloud Storage is a flexible, fully managed object storage service by Google Cloud.

Azure Blob Storage
Azure Blob Storage is Microsoft's object storage solution in the Azure cloud.

Most of these common implementations offer the same features, so choosing one over the others is more a matter of developer familiarity than anything.

Cache

Laying the Groundwork

Whenever you're fetching data from a database, it takes time and compute resources to do. While each request might only take milliseconds, system design is all about designing for scale, and with millions of users, this can lead to significant problems, such as:

- High Latency: Slow responses frustrate users and make your app feel sluggish
- Low Availability: A flood of requests can overwhelm your database or API, leading to reduced performance or even downtime
- Poor Efficiency: Fetching the same data repeatedly wastes valuable compute resources

One of the most common kinds of systems is one in which there is a high volume of reads, but the data doesn't update very often - we call this a **high-read, low-write system**. Think of social media apps, where users spend the majority of their time reading other posts, and only make their own posts occasionally. Or e-commerce platforms, where products get viewed millions of times, but rarely have their details changed.

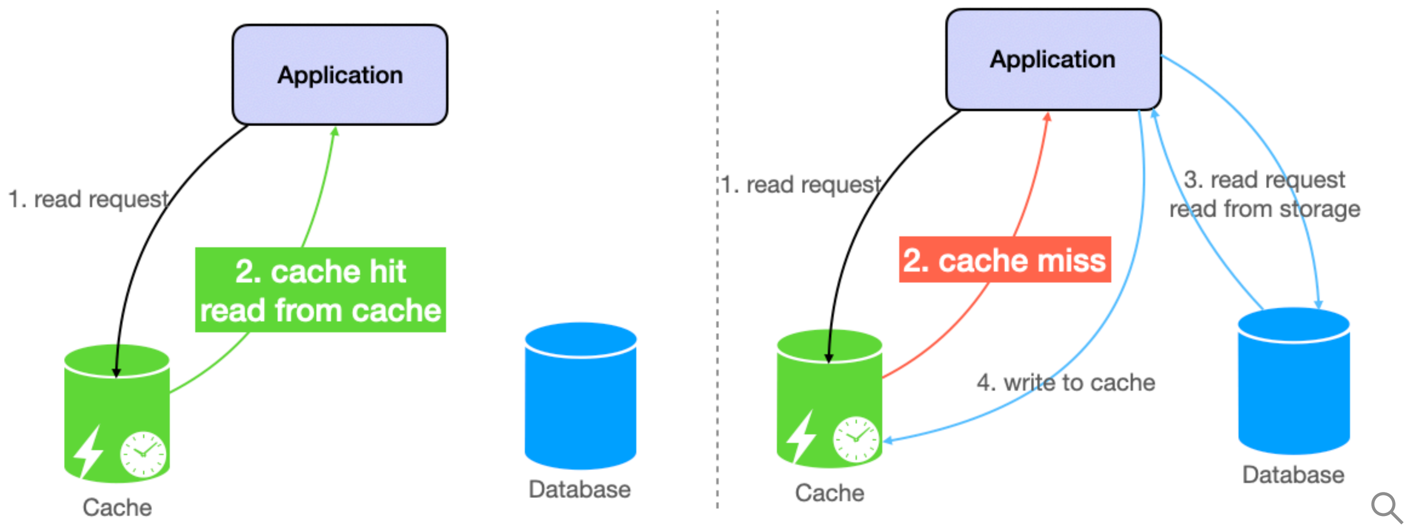
If you think it seems a bit redundant to be constantly fetching the same, unchanged data from a database, you're right! Why do we need to go all the way to the database if we've already fetched something, can't we store it closer to us for next time?

This is where **caches** come in.

Technical Explanation

At a high level, a cache works like this:

1. A request comes in. Before heading to the database or API, your system checks if the data is already in the cache.
2. If the data is in the cache, which we call a **cache hit**, it gets served immediately - no need to bother the database.
3. If the data isn't in the cache, which we call a **cache miss**, your system fetches it from the database like usual... but this time, it also saves a copy in the cache for next time.



By keeping frequently-used data close at hand, caches reduce the time it takes to respond to requests and lessen the load on your backend systems. This improves latency, boosts availability, and optimizes efficiency, making your app more scalable and responsive.

In an interview, you'll also be expected to be able to explain the following concepts:

- **When to remove items from cache**: An eviction policy is a set of rules that determines which data to remove from the cache when it reaches its capacity. Some common eviction policies are:
  - Least Recently Used (LRU): Least recently used items are removed first.
  - First In, First Out (FIFO): Items are removed in the order they were added.
  - Least Frequently Used (LFU): Least frequently used items are removed first.
- **How to keep cache data up to date:** An invalidation strategy is a set of rules that determines how and when cached data is marked as stale or removed to keep it consistent with the source data. Some common invalidation strategies are:

  - Time-To-Live (TTL): Data is removed after a pre-set expiration time.
  - Event-Based Invalidation: The cache is cleared or updated when the underlying data is updated.
  - Manual Invalidation: Cache entries are explicitly removed or refreshed by the application.
- **How to store data in the cache**: A cache write strategy is a set of rules that determines how and when data is written to the cache to ensure it is available for future requests. Some common cache write strategies are:
  - Write-Through: Data is written to the cache and the underlying database simultaneously, keeping them in sync.
  - Write-Behind: Data is first written to the cache and then asynchronously written to the database, improving write performance.
  - Write-Around: Data is written directly to the database and added to the cache only when it is read, reducing cache pollution for infrequently accessed data.

Common Implementations

1. In-Memory vs. Disk-Based Caches

Caching systems can be categorized based on where the data is stored: either in memory or on disk.

In-Memory Caches store data in RAM, providing extremely fast access.

Key Features:

- Low latency (microseconds to milliseconds)
- Volatile storage: data is lost if the cache is restarted (unless persistence is enabled)

When to Use:

- For real-time applications requiring high-speed access, such as session management, leaderboards, or frequently accessed database queries

Examples: Redis, Memcached

Disk-Based Caches store data on persistent storage like hard drives or SSDs, offering slower but more durable caching.

Key Features:

- Data persists through restarts, making it ideal for long-term storage of cacheable data
- Slower than in-memory caches due to disk I/O latency

When to Use:

- For caching large datasets or static assets, where persistence is critical and access speed is less important

Examples: Varnish Cache, browser caches

2. Client-Side vs. Server-Side Caches

Caching systems can also be categorized based on where the cache is located within the architecture.

Client-Side Caches store data on the user's device (e.g. in a browser or app).

Key Features:

- Reduces server communication by caching data locally
- Specific to individual users, enabling faster responses for their repeated requests

When to Use:

- For static assets (e.g., images, CSS, JavaScript) or user-specific data in offline-capable applications

Examples: Browser Cache, LocalStorage, IndexedDB

Server-Side Caches store data on or near the server, shared across all users and requests.

Key Features:

- Reduces load on backend systems like databases and APIs
- Optimizes performance for high-traffic applications by serving shared data quickly

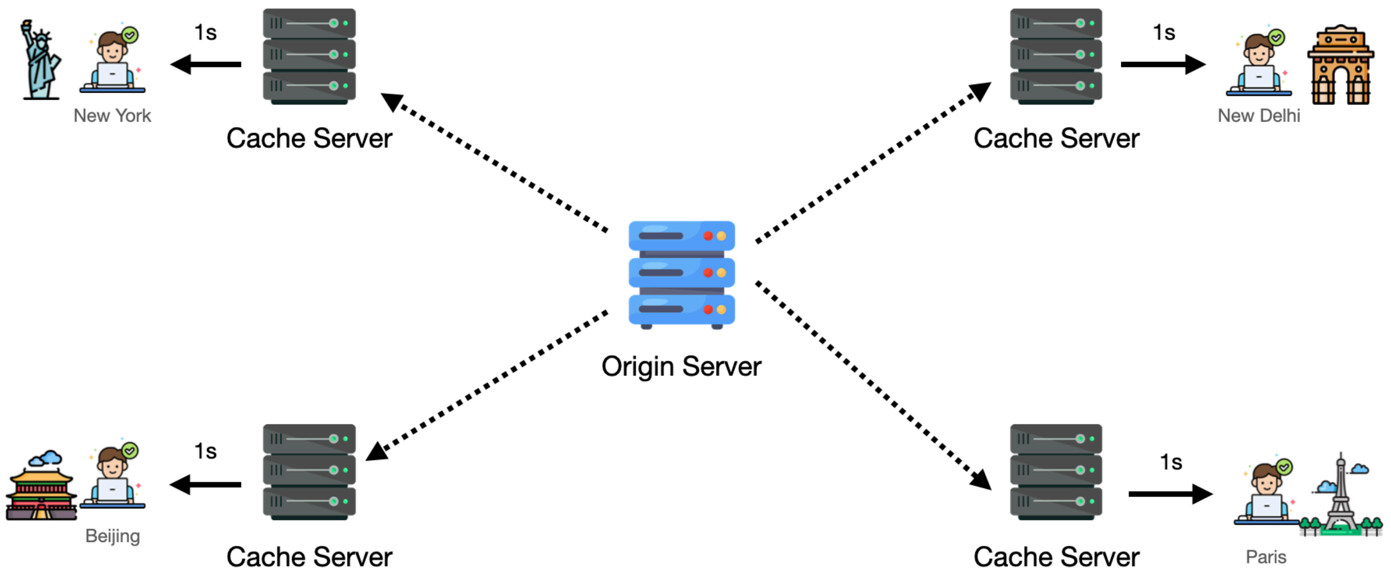
When to Use:

- For frequently accessed data that is consistent across users, such as API responses, product pages, or popular posts

Examples: Redis, CDNs, NGINX Cache

CDN (Content Delivery Network)

Laying the Groundwork
Ever noticed how some websites load almost instantly, even when they have heavy images, videos, or other assets? That's often thanks to a special type of server-side cache, known as a **CDN**, or, **content delivery network**. A CDN is a distributed network of servers that helps deliver content to users more quickly and efficiently by storing cached copies of static assets closer to the user's location.



The key challenges a CDN solves include:

- High Latency: When users are far from your server, it takes longer for requests and responses to travel back and forth, causing delays.
- Bandwidth Overload: A surge in traffic to your website can overwhelm your origin server, leading to slowdowns or crashes.
- Global Scalability: Hosting all your assets in a single location makes it hard to serve users worldwide without performance issues.

Static assets like images, videos, JavaScript files, and even API responses are cached across multiple servers globally, so users get their content from a server that's physically closer to them. This reduces latency, distributes traffic more evenly, and keeps your origin servers from being overburdened.

Technical Explanation

You've already learned how caches work, and CDNs are essentially just a common cache implementation. But still, it's good to go over how they work specifically:

- Content Distribution: When you deploy your app, static files (e.g. images, CSS, videos) are uploaded to your CDN provider. The CDN copies these files to its network of edge servers located around the globe
- Request Handling: When a user visits your site, their request is routed to the nearest CDN server
  - If the requested file is already cached there (cache hit), the server delivers it immediately
  - If the file isn't cached (cache miss), the CDN fetches it from the origin server, caches it locally, and then serves it to the user
- Caching Strategy: CDNs use caching policies like TTL to determine how long files should be stored before refreshing from the origin server.

Common Implementations

Cloudflare

Cloudflare is a popular CDN known for its ease of use and strong security features.

Key Features:

- Global network with low latency
- Built-in DDoS protection and Web Application Firewall
- Ideal for web applications needing speed and security with minimal configuration

AWS CloudFront

AWS CloudFront is Amazon's fully managed CDN integrated with AWS services.

Key Features:

- Seamless integration with AWS storage (S3) and compute (Lambda)
- Supports dynamic and static content delivery
- Ideal for applications already hosted in AWS or requiring custom logic at the edge

Akamai

Akamai is one of the oldest and most robust CDNs, typically used by enterprise-level clients.

Key Features:

- Industry-leading global server network for ultra-low latency
- Advanced customization and analytics tools
- Ideal for enterprise applications with high traffic and advanced delivery requirements

Message Queues

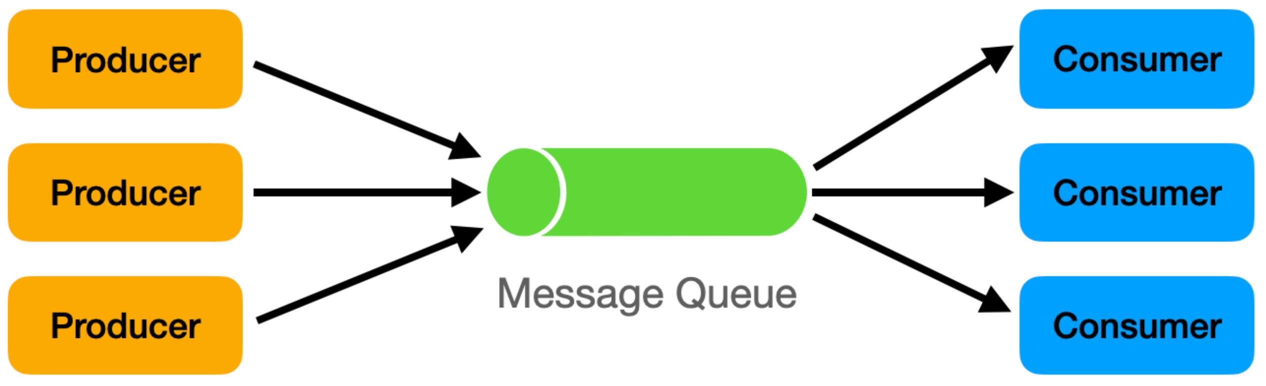
Laying the Groundwork

When building modern distributed systems, one of the most common challenges is communication between different services. Systems often need to send and process large volumes of tasks or data while staying reliable and scalable. However, directly connecting services can lead to several problems:

- Tight Coupling: Services become dependent on each other, making the system harder to scale or modify
- Overloading: If one service generates more tasks than another can handle, it can lead to failures or bottlenecks
- Task Management: Without a way to track tasks, it's easy to lose or duplicate data when services crash or restart

For example, in an e-commerce system, the order service might need to notify the inventory service, payment service, and shipping service. If all these interactions happen directly, any failure or delay could break the entire system.

A **message queue** solves these problems by acting as an intermediary. It allows services to send messages without worrying about whether the receiving service is ready to process them. This makes systems more reliable, decoupled, and scalable.



Technical Explanation

At a high level, a message queue works like this:

1. Producers send messages (tasks or data) to the queue. A producer could be any service generating work, such as an order service in an e-commerce platform.
2. The queue temporarily stores these messages until they are processed. Messages are stored in the order they arrive.
3. Consumers retrieve messages from the queue and process them. For example, a payment service might consume a message about a new order to initiate a payment.

This pattern works well because it decouples producers and consumers. Producers don't need to wait for consumers to process tasks; they just send messages and move on. And consumers can process messages at their own pace, making the system more resilient to spikes in load or partial failures.

In an interview, you should also be able to explain the following key concepts:

- Acknowledgements: Consumers send an acknowledgment to the queue after successfully processing a message. If no acknowledgment is received, the message can be re-delivered to ensure reliability.
- Dead Letter Queues (DLQs): Messages that fail to be processed repeatedly are sent to a separate "dead letter" queue for debugging or manual handling.
- Message Ordering: Some queues ensure messages are delivered in the same order they were produced (FIFO), while others allow out-of-order processing for higher throughput.

Common Implementations

1. Message Queue Types

Point-to-Point (P2P) is a type of message queue where a single consumer processes each message.

Example: Processing user orders in an e-commerce system

Tools: RabbitMQ, AWS SQS

Publisher-Subscriber (Pub/Sub) is another type of message queue where multiple consumers can subscribe to a topic and receive messages.

Example: Sending order updates to inventory, payment, and shipping services simultaneously

Tools: Apache Kafka, Google Pub/Sub

2. Key Providers

RabbitMQ is a widely-used message broker that supports both P2P and Pub/Sub models. It is best used for traditional queueing with complex routing and acknowledgment features.

Apache Kafka is also widely-used, and is a distributed event streaming platform designed for high-throughput use cases. It is particularly ideal for Pub/Sub scenarios and real-time analytics.

AWS SQS (Simple Queue Service) is a fully managed, scalable message queue service. It is great for cloud-based systems needing P2P messaging with minimal setup.

API Gateway

Laying the Groundwork

When building modern distributed systems, especially those using microservices, managing how clients interact with your backend becomes a critical challenge. APIs are the bridges connecting clients (like web browsers or mobile apps) to backend services, but when you have dozens, or even hundreds, of microservices, exposing each one directly to clients can lead to several problems:

- Inconsistent Interfaces: Different services might have varying API standards, making it difficult for clients to interact seamlessly
- Increased Latency: Clients may need to make multiple calls to different services, leading to slower response times
- Security Concerns: Exposing all your services directly to clients increases the attack surface, making it harder to enforce consistent security measures
- Traffic Spikes: Backend services might get overwhelmed by a sudden influx of requests, leading to potential downtime

An **API Gateway** solves these issues, by acting as a single entry point for all API requests. It routes requests to the appropriate services, handles cross-cutting concerns like authentication, and even helps optimize performance with caching and rate limiting. Think of it as a traffic cop that directs and controls the flow of requests in your system.

Technical Explanation

At a high level, an API Gateway works like this:

1. Request Handling: Clients send API requests to the gateway instead of directly to backend services
2. Routing: The gateway then inspects the request and determines which backend service it needs to forward the request to
3. Cross-Cutting Features: While processing requests, the gateway can perform a variety of tasks:
   - Authentication: Ensures the request comes from a valid user or system
   - Rate Limiting: Caps the number of requests a client can make in a specific time frame to protect backend services
   - Caching: Serves frequent requests from a cache instead of hitting backend services, reducing latency
   - Logging and Monitoring: Tracks request details for debugging or usage analytics
4. Response Aggregation: For some use cases, the gateway might fetch data from multiple backend services and combine the responses into a single payload for the client

API Gateways not only simplify how clients interact with backend services, but also centralize management for features like security, traffic control, and monitoring, making systems easier to scale and maintain.

In an interview, you may also need to discuss these key concepts:

- Authentication and Authorization: Explain how API Gateways handle tokens (like OAuth or JWT) to ensure secure access
- Rate Limiting: Demonstrate how to prevent abuse or overload by capping request rates
- Load Balancing: Show how gateways distribute incoming requests across multiple instances to optimize performance and reliability

Common Implementations

AWS API Gateway

AWS API Gateway is a fully managed API gateway provided by AWS.

Key Features:

- Integrates seamlessly with AWS services like Lambda and DynamoDB
- Built-in support for caching, authorization, and throttling
- Ideal for serverless or cloud-native architectures requiring minimal setup

Kong Gateway
Kong Gateway is an open-source, extensible API gateway.

Key Features:

- Plugin architecture for adding custom features
- High-performance routing and load balancing
- Ideal when flexibility and open-source tooling are needed

NGINX API Gateway
NGINX API Gateway is a lightweight, high-performance API gateway based on NGINX.

Key Features:

- Combines API gateway functionality with reverse proxy and load balancing
- Ideal for high-throughput, low-latency applications requiring minimal overhead

Interview Step-by-Step

The key to success in system design interviews is following a structured approach. As we learn this interview process, we'll actually show you how to solve a popular system design problem, Design Twitter, so you can see how it works in a real interview!

Functional Requirements

At the start of any system design interview, the first step is to define your core functional requirements. This stage usually takes only a few minutes.

To do this, you should think of requirements in terms of "Users can do...".

So, for example, for an app like Spotify, what can users do? Well, they can listen to songs, they can make playlists, and they can also upload their own songs. Just like that, you've defined some of the core functional requirements.

Remember that your interview is likely only 45-60 minutes. You don't have to cover every single requirement - instead, your interviewer wants to see that you can come up with some of the core/most important things that users can do with the system.

Let's take our Twitter example that we're walking through. Think to yourself for a moment: What are some things that users can do on Twitter?

For the system we will design during this course, we came up with these ones:

- Users need to be able to post tweets
- Users need to be able to view individual tweets
- Users need to be able to view feed
- Users need to be able to follow other users
- Users need to be able to like tweets
- Users need to be able to comment on tweets

Non-Functional Requirements

After defining functional requirements, we will move on to non-functional requirements. This stage also usually only takes a few minutes.

While functional requirements outline what the user can do, non-functional requirements outline what the system should do/have in order to support those functional requirements.

Common considerations include:

- Performance: How fast does the system need to respond to user actions or process requests? This often includes defining acceptable latency thresholds for key operations like loading a user's feed or posting a tweet.
- Availability: How consistently should the system be accessible to users? This could mean ensuring near 100% uptime through redundancy, failover mechanisms, and proactive monitoring.
- Scalability: Can the system handle increasing numbers of users, data, or traffic without a degradation in performance? Scalability can be both vertical (upgrading resources on a single server) and horizontal (adding more servers or instances). For large-scale systems, horizontal scalability is often preferred.
- Reliability: How well does the system handle failures? This includes designing for fault tolerance so that the system can recover from crashes or unexpected errors without losing functionality or data.
- Consistency: How accurately does the system reflect the same data across all users and operations? In systems with distributed databases, maintaining strong consistency can be challenging but critical for certain operations.
- Durability: How safely is data stored? This involves ensuring that data, once written, isn't lost due to hardware failures, power outages, or other disruptions.
- Security: How well does the system protect against unauthorized access or attacks? This includes safeguarding sensitive user data and preventing exploits such as SQL injection or distributed denial-of-service (DDoS) attacks.

Let's take a look again at our Twitter example. We know that our functional requirements are these:

- Users need to be able to post tweets
- Users need to be able to view individual tweets
- Users need to be able to view feed
- Users need to be able to follow other users
- Users need to be able to like tweets
- Users need to be able to comment on tweets

So, what characteristics should our system have? Well, here are the basic ones we came up with:

- Low Latency: Ensure users can post tweets and view their feed within 200 milliseconds to provide a smooth experience

- High Availability: Maintain as close to 100% uptime to ensure the platform is always accessible to users
- Scalability: Support horizontal scaling to handle growth in the user base and increased traffic seamlessly
- Data Durability: Store tweets, likes, and comments in a distributed, fault-tolerant system to prevent data loss

API Design

After defining our requirements, it's time to move on to the API design stage. Again, this stage should only take a few minutes.

A lot of people really struggle at this part simply because they overcomplicate it. Ultimately, all this stage is is just turning your functional requirements into API endpoints. To keep it simple, there should be one endpoint per functional requirement.

In this section, interviewers are looking for three things:

1. Readable paths: You should pick easily understandable names. For an endpoint that deals with Tweets, /tweet makes a lot more sense than /item or something - this is pretty common sense but you'd be surprised how many people choose confusing names.
2. Data types: You should know what data is being sent and received with each API. For an endpoint that creates new Tweets, you might be sending a user ID (integer) and some Tweet content (string), and you might be receiving a Tweet ID (integer) and maybe some response codes. This isn't a coding interview, so keep it simple.
3. HTTP methods: You should know what HTTP methods are used with each endpoint. For an endpoint that creates new Tweets, you would use a POST method, whereas an endpoint that fetches Tweets would be a GET method.

For our Twitter walkthrough, this is what we came up with:

Users need to be able to post tweets:

```
POST /tweet
Request
{
  “user_id”: “string”,
  "content": "string"
}
Response
{
  "tweet_id": "string",
  "status": "string"
}
```

Users need to be able to view individual tweets:

```
GET /tweet/<id>
Response
{
  "tweet_id": "string",
  "user_id": "string",
  "content": "string",
  "likes": "integer",
  "comments": "integer"
}
```

Users need to be able to view feed:

```
GET /feed
Response
[
  {
    "tweet_id": "string",
    "user_id": "string",
    "content": "string",
    "likes": "integer",
    "comments": "integer"
  }
]
```

Users need to be able to follow other users:

```
POST /follow
Request
{
  "follower_id": "string",
  "followee_id": "string"
}
Response
{
  "status": "string"
}
```

Users need to be able to like tweets:

```
POST /tweet/like
Request
{
  "tweet_id": "string",
  "user_id": "string"
}
Response
{
  "status": "string"
}
```

Users need to be able to comment on tweets:

```
POST /tweet/comment
Request
{
  "tweet_id": "string",
  "user_id": "string",
  "comment": "string"
}
Response
{
  "comment_id": "string",
  "status": "string"
}
```

High-Level Design

High-level design is the bulk of the interview, and is where you'll combine the work from the first few parts with your knowledge of system design. This should take around 15-20 minutes, but varies based on your level (juniors can expect to spend more time on this, whereas seniors will likely be expected to move to deep dives more quickly).

High-level design is what stumps the most people, because they often don't know where to even begin. Again, luckily for you, we have a tried and tested framework for easily building up to a working system.

You should start by addressing your functional requirements in the most basic way possible, and then after you have a feasible solution, start addressing the non-functional requirements by making it more robust.

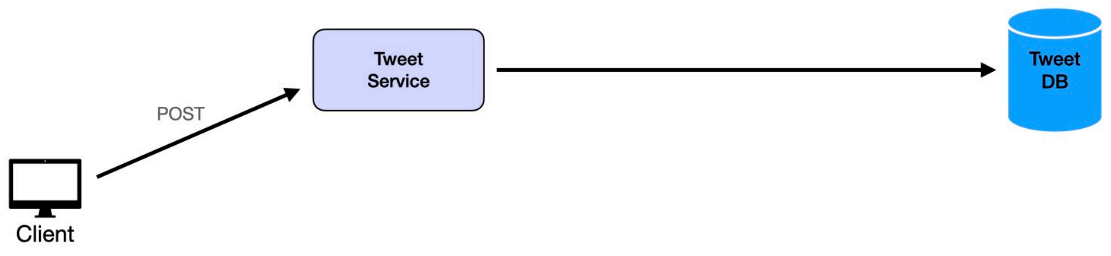
Functional Requirements

To start with the functional requirements, you're just going to take your API endpoints and map the data flow with some services. Your goal is to show that your thinking is very methodical and structured, so it's best to start simple without trying to add any complicated parts yet - this way, if you've made a mistake, the interviewer can correct it early on.

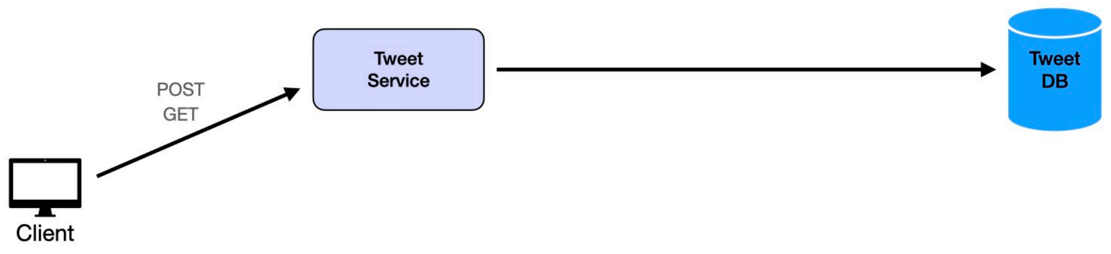
- Wondering why microservices is the best approach? Check out this: {here}

So, take a look at how we do this for the Twitter example:

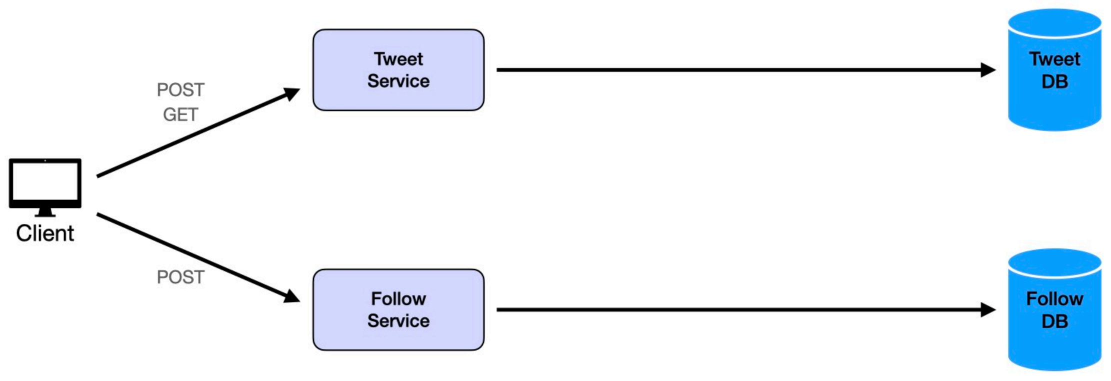
Users need to be able to post tweets:



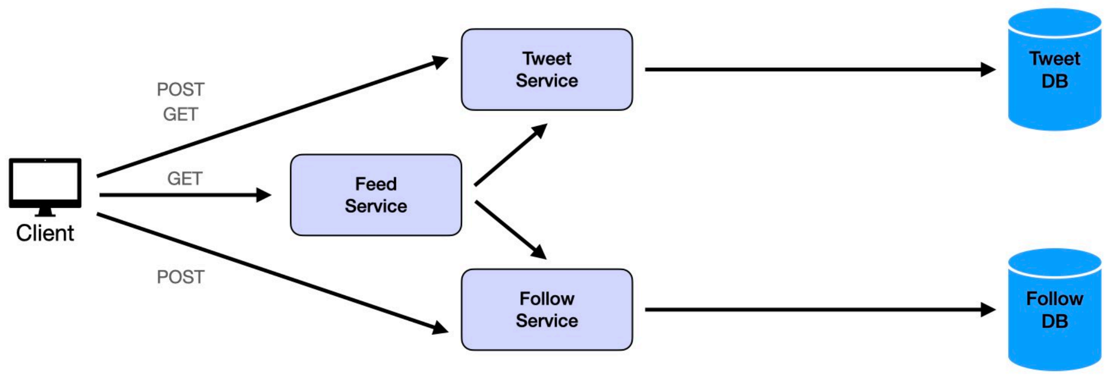
Users need to be able to view individual tweets:



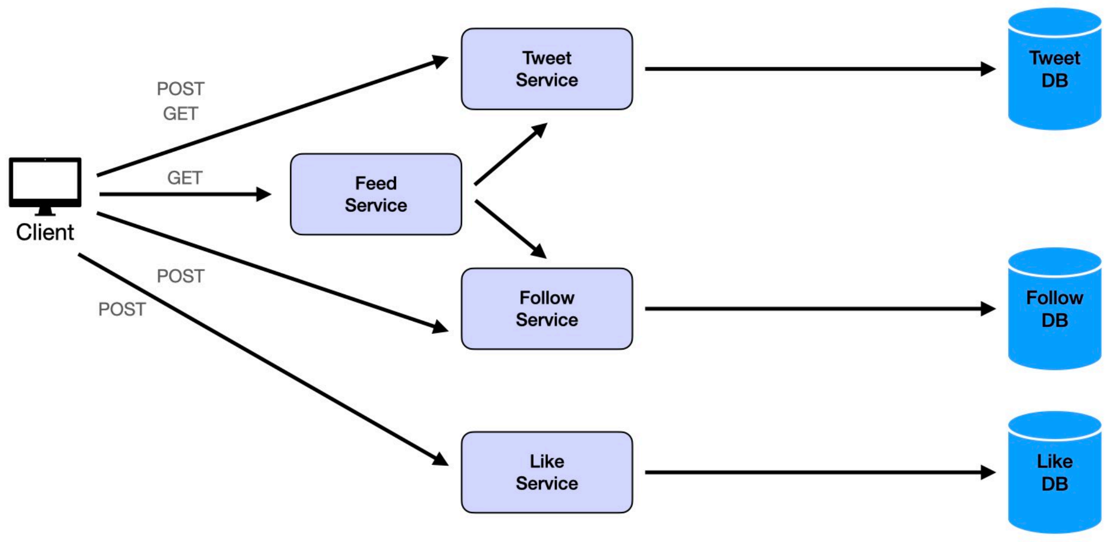
Users need to be able to follow other users:



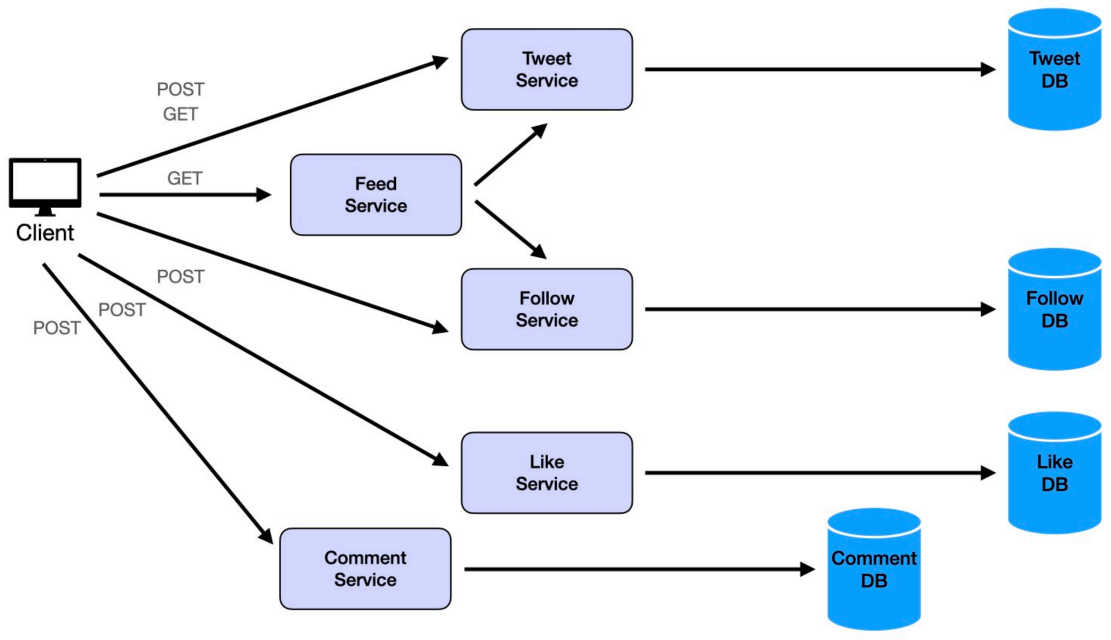
Users need to be able to view feed:



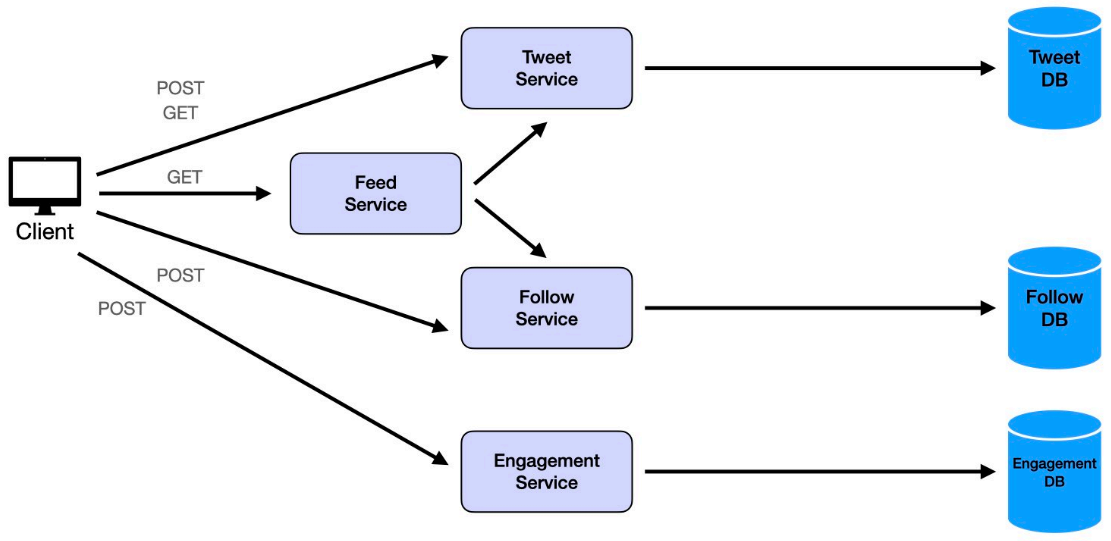
Users need to be able to like tweets



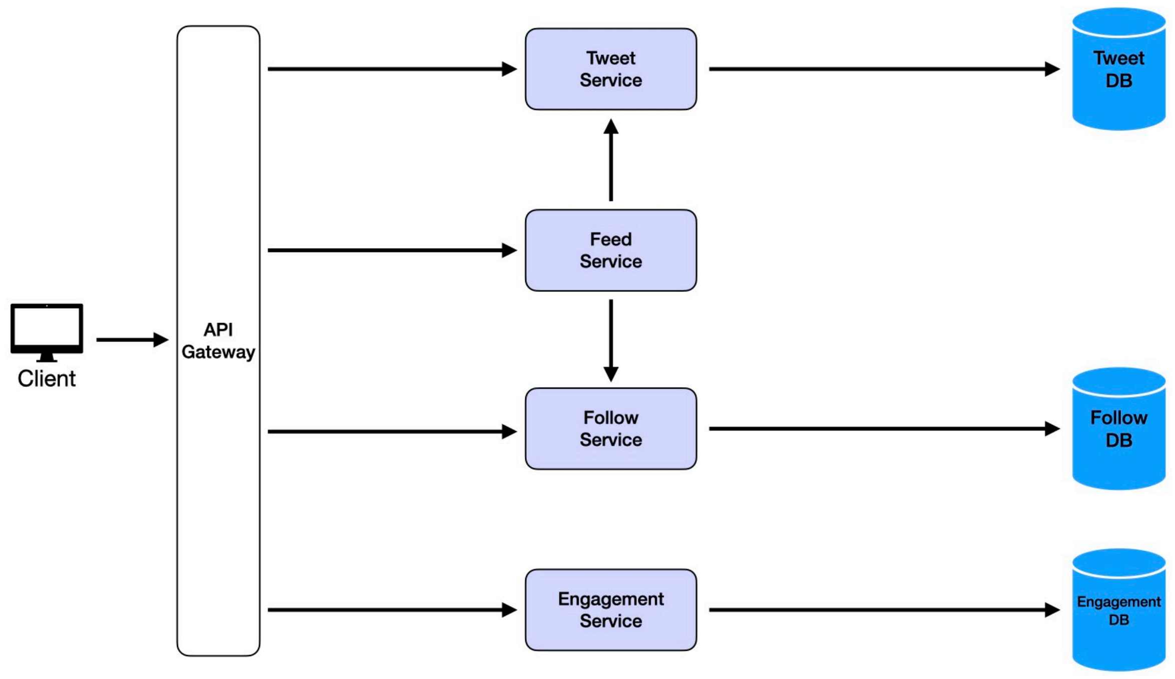
Users need to be able to comment on tweets:



However, this is a lot of services, and even though a separation of concerns is important, we don't want to go too overboard and turn everything into a microservice. One thing we want to think of is, is there any way we can combine some services? In this case, I'd argue that likes and comments are logically very similar - both are ways of engaging with tweets, both are a type of counter on a tweet, they just contain different data. Well, this makes them very similar, and in this case, we should consider merging them into a single Engagement Service and Engagement Database.



Now lastly, with multiple services, it's almost always a good rule of thumb to add in an API gateway (make sure to go back to the above section, Main Components, if you're unsure what that is).

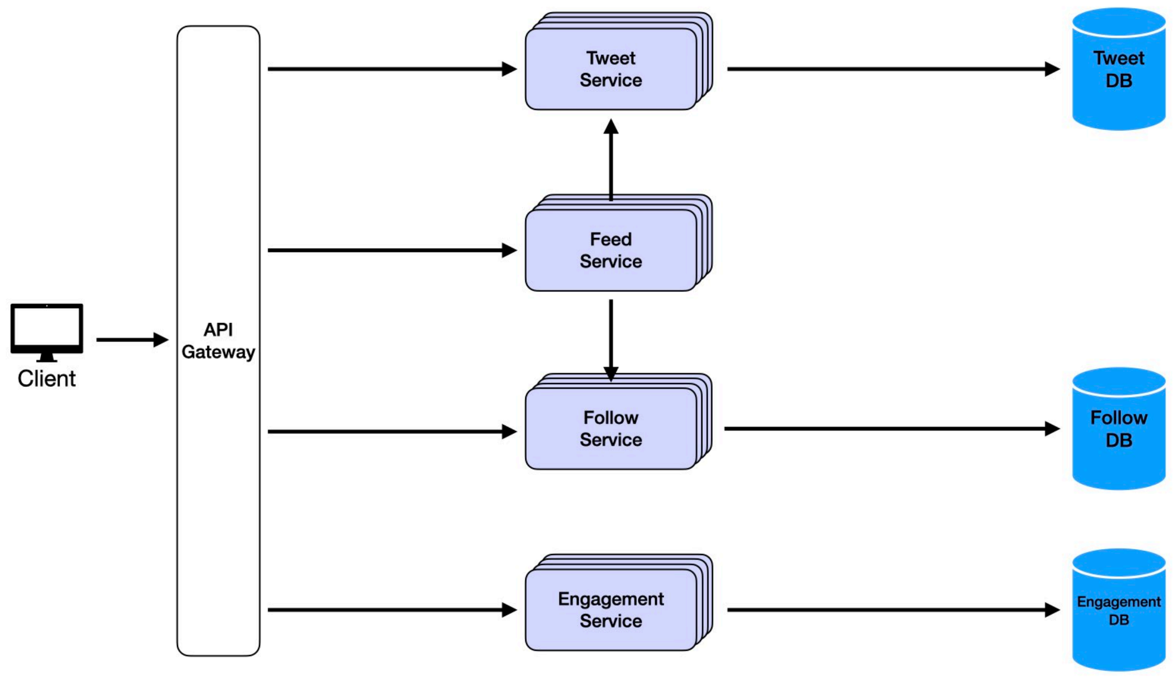


Non-Functional Requirements

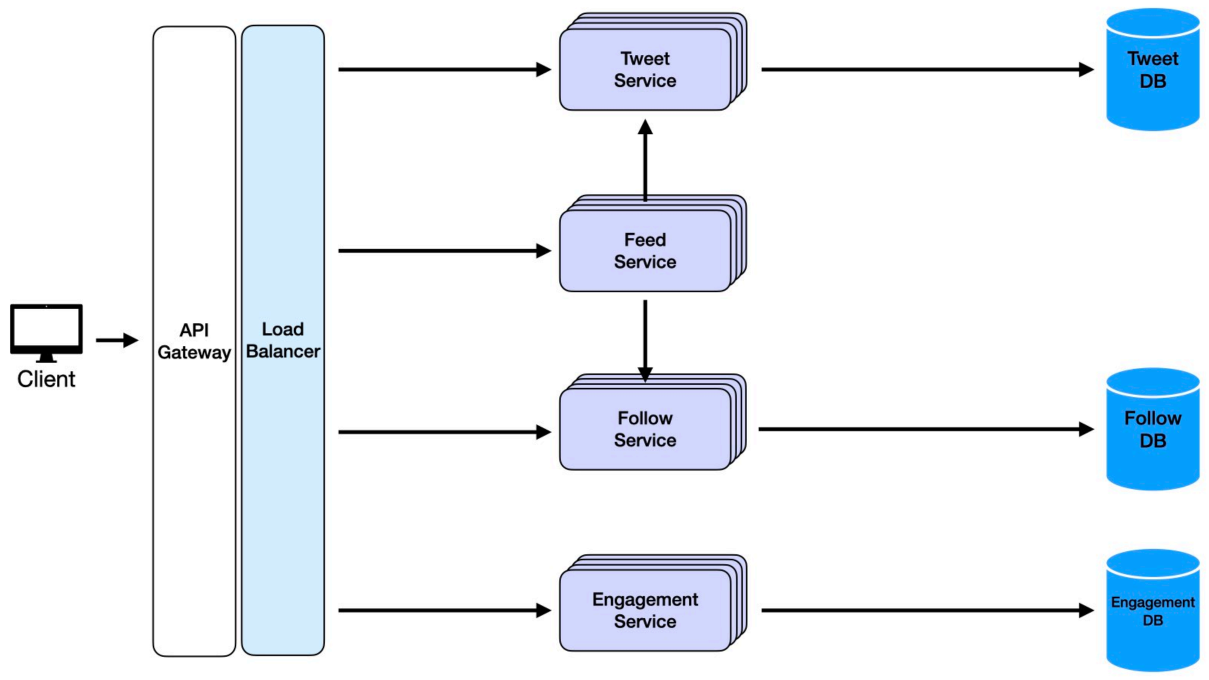
Great, now we have a structured system that is at least feasible for solving the functional requirements! However, this system clearly has significant challenges at scale, and doesn't really show any of our advanced system design knowledge. Now let's move into part 2 of high-level design: building for the non-functional requirements.

Let's address each non-functional requirement one step at a time. The biggest mistake candidates make here is not tying these recommendations back to the non-functional requirements they called out earlier in the interview.

We'll start with **scalability**, as it's a relatively straightforward component to address in most cases. As outlined in the non-functional requirements section, our goal is to support horizontal scaling to handle growth in the user base and increased traffic seamlessly. Horizontal scaling allows us to add more instances of our services to handle higher loads, ensuring consistent performance as demand increases.



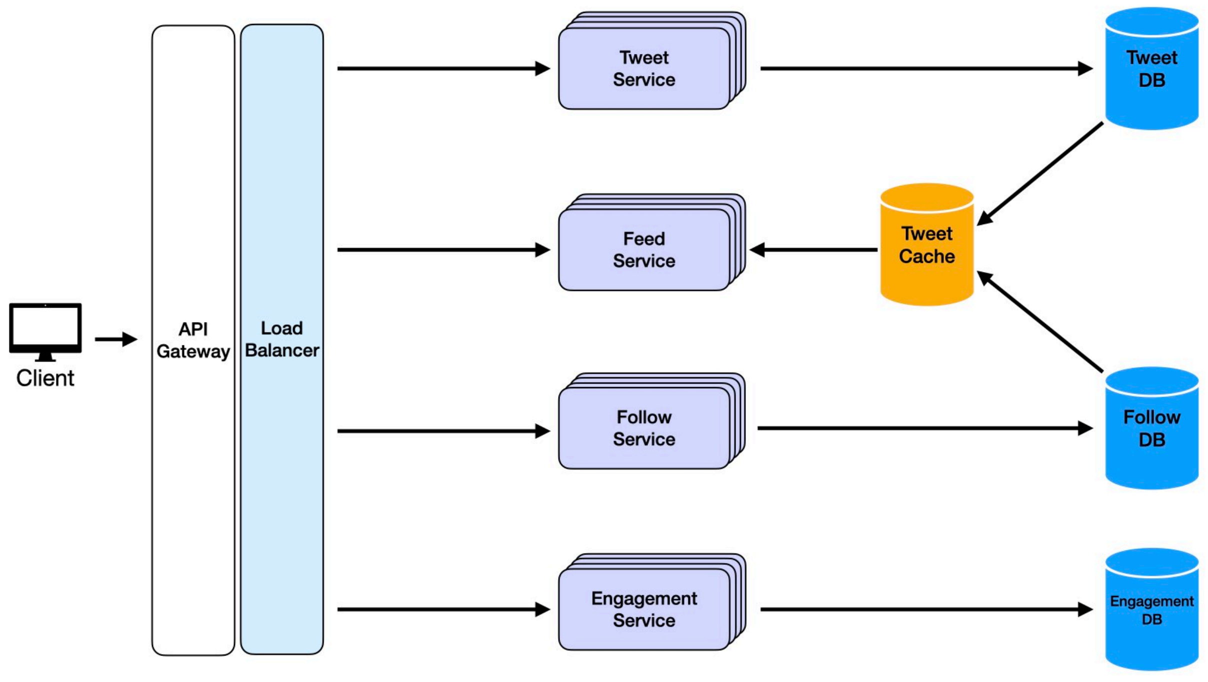
To implement horizontal scaling, we will deploy multiple instances of each service - Tweet Service, Feed Service, Follow Service, and Engagement Service. These instances will operate independently, handling requests in parallel. However, to distribute traffic efficiently across these instances, we need a load balancer.



A load balancer ensures incoming requests are evenly distributed across available service instances. It also performs health checks to monitor the status of each instance and reroutes traffic away from unhealthy instances, ensuring high availability. This is actually one of our other non-functional requirements that we'll get to in a moment, and is a nice bonus we get from this load balancer. By incorporating a load balancer, we can scale each service dynamically based on traffic patterns. For example, during peak hours, more instances of the Feed Service can be spun up to handle the surge in requests, and these can scale back down during periods of lower activity to optimize resource usage.

Now let's move on to another core non-functional requirement: low latency. When we have only a few users posting tweets, our servers should run blazing-fast, as each user's feed only fetches a handful of tweets. But as we start to scale to millions of users and billions of tweets, accessing the database for every feed request will start to become exponentially slow with our current system. It seems a bit redundant for users to write tweets to the database, and then for the feed to re-pull these from the database each time, no?

This is where a cache comes to the rescue!

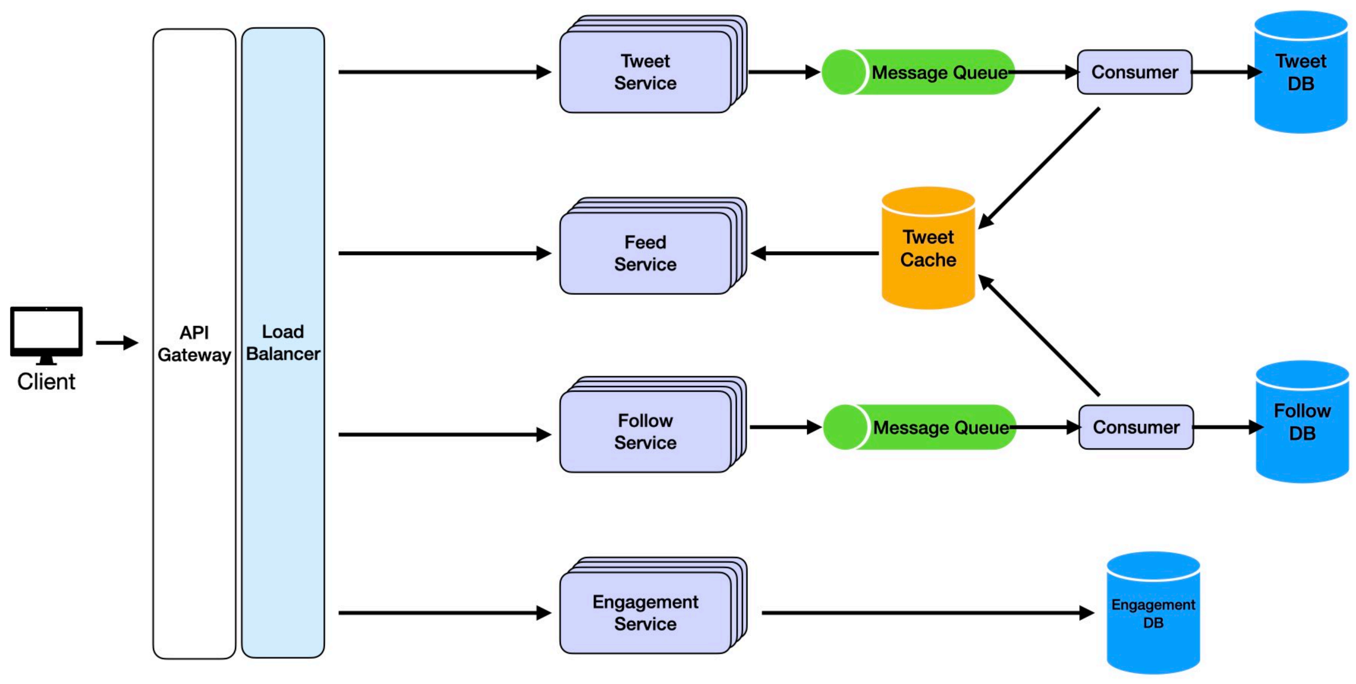


A cache is a high-speed data storage layer that stores frequently accessed data closer to the application. In our system, the Feed Service could leverage a distributed caching system like Redis or Memcached to store the most recent tweets for each user. Here’s how it works:

1. Feed Precomputation:
   - When a user posts a tweet, the Feed Service doesn’t just update the followers’ feeds in the database. It also pushes the new tweet to a cache, storing it as part of the precomputed feeds for the user’s followers.
   - This way, when followers log in, the Feed Service can fetch the feed data directly from the cache instead of querying the database or relying on real-time aggregation.
2. Hot Data Access:
   - Caches are ideal for storing "hot" data - data that is accessed frequently, such as the latest tweets for a user's feed. Since caches operate in memory, they can deliver this data in milliseconds, reducing response times and improving the user experience.
3. Reducing Database Load:
   - By offloading repeated reads to the cache, we reduce the load on the database. This makes the system more scalable and ensures that the database is available for other critical write operations.

4. Cache Expiry and Consistency:
   - To ensure the cache stays fresh, we can set an expiry time for cached items or use an event-driven update model. For example, when a new tweet is posted, an event triggers the cache to update, ensuring followers see the latest tweets without unnecessary delays.

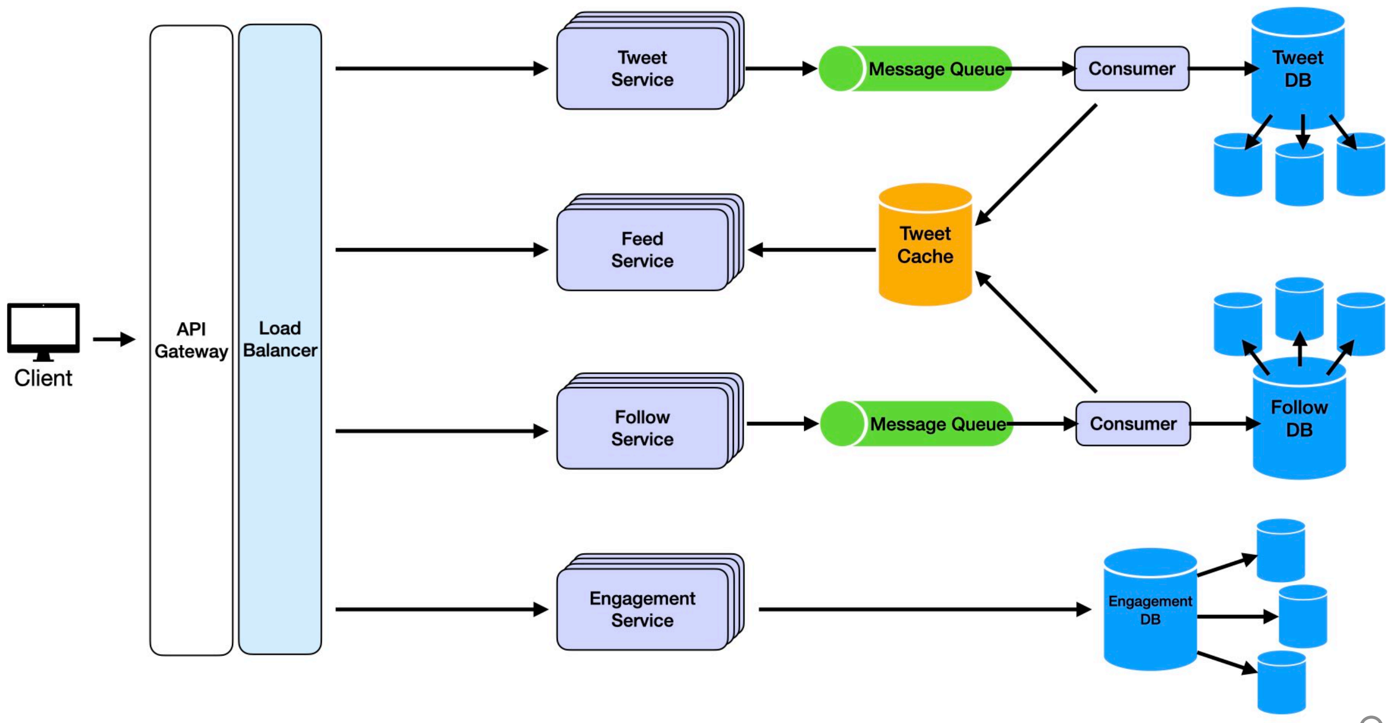
Now, when adding in the load balancer earlier, we touched on how they have an additional benefit, which is that the health checks help us maintain high availability. But what else can we do to have **high availability**? Well, let me present you with a situation where we might not have availability with the current system. What happens if a user posts a tweet at 1:02pm, but at 1:03pm, our tweet service goes down, and at 1:04pm, their followers log onto the app. Are their followers going to see this tweet in their feed? As it stands now, NO! Now here's another scenario. What happens when we have millions of active users publishing tweets all at the same time? If we tried to process them all simultaneously, we would overload our servers! Hmm, if only there was a solution to this... wait a minute, this is exactly why we have message queues!



A message queue acts as a buffer between services, decoupling their dependencies and ensuring that messages (like new tweets) are not lost, even if one of the services experiences downtime. Here's how it works: when a user posts a tweet, the Tweet Service doesn’t directly communicate with the Feed Service. Instead, it places the tweet in a queue, which is then processed by a consumer and added to both the database and cache. The Feed Service, which processes tweets to update users' feeds, then reads from this cache. This way, even if the Tweet Service goes offline, the messages (tweets) are safely stored in the queue and processed by the consumers, and the Feed Service can still update the feeds of the user’s followers when they log in.

By incorporating a message queue, we ensure eventual consistency and high availability even during partial system failures. In our scenario, followers would still see the tweet in their feed, thanks to the queue ensuring that no message is lost. This decoupling of services also helps out with scalability, as the queue can handle varying workloads and traffic spikes without overwhelming downstream services. Message queues like RabbitMQ, Kafka, or AWS SQS are built for durability and reliability, making them a perfect fit for our use case.

While we've addressed a lot of the service non-functional requirements, there is still one last critical requirement to tackle: data durability. In a system that handles billions of tweets, likes, and follows, ensuring that data is never lost is essential, because once lost, we cannot get it back. How do we prevent this from happening? This is where we take our databases a step further and use a distributed databases model.



A distributed database is designed to replicate and store data across multiple nodes in a cluster. Distributed databases like Amazon DynamoDB, Google Cloud Spanner, or Cassandra automatically replicate data across multiple nodes. This means that even if one node goes down, the data is still accessible from other replicas. Additionally, distributed databases provide built-in mechanisms for point-in-time recovery and automated backups. In this case, regular snapshots of the Tweet DB, Follow DB, and Engagement DB can be taken and stored in a separate backup system. And, if we ever have a complete failure, the data can be restored to its last consistent state.

Deep Dives

Deep dives are the last step of the system design interview, and usually focus on addressing higher-level, more specific challenges or edge cases in your system. They go beyond the high-level architecture to test your understanding of advanced features, domain-specific scenarios, and trade-offs.

1. How would you handle the Celebrity Problem?

The "celebrity problem" arises when a user with millions of followers posts a tweet, creating a massive fan-out as their tweet needs to be added to millions of follower feeds. This can overwhelm the system and lead to latency and high write amplification.

Solution: We can modify our existing architecture to handle this more efficiently:

- Fan-Out-on-Write for Normal Users: For users with a manageable number of followers, we continue with the standard fan-out-on-write model. The tweet is pushed to their followers' feeds as soon as it's posted.
- Fan-Out-on-Read for Celebrities: For users with a large follower count (e.g., more than 10,000), we switch to a fan-out-on-read model. The celebrity's tweets are stored in the Tweet Database and Tweet Cache but not precomputed into individual follower feeds. When a follower opens their feed, the Feed Service dynamically fetches the celebrity's latest tweets from the cache or database and merges them into the user's timeline.
- Dynamic Switching: Implement a threshold (e.g., follower count or engagement volume) to dynamically decide whether to use fan-out-on-write or fan-out-on-read for a given user.

2. How would you efficiently support Trends and Hashtags?

Twitter trends and hashtags involve aggregating data across billions of tweets in real-time to identify popular topics. How can we compute and update trends efficiently?

Solution: We enhance the existing architecture as follows:

- Distributed Trend Computation: Each region or data center computes local trends by aggregating hashtags and keywords using a sliding window algorithm (e.g., the past 15 minutes). Local results are sent to a global aggregation service, which combines them to generate global trends.
- Hashtag Indexing: Modify the Tweet Service to index hashtags upon tweet creation: Maintain an inverted index where hashtags are keys, and associated tweet IDs are values. Use a distributed search engine like Elasticsearch or Solr to store and query the hashtag index efficiently.
- Caching Trends: Trends are calculated periodically (e.g., every minute) and cached in a distributed cache like Redis for low-latency access. A TTL (time-to-live) ensures trends are refreshed frequently without overwhelming the system.

3. How would you handle Tweet Search at Scale?

Search is a core feature of Twitter, allowing users to search tweets, hashtags, and profiles. How can we support a scalable, real-time search system?

Solution: We incorporate a distributed search architecture:

- Real-Time Indexing: Modify the Tweet Service to send newly created tweets to a search indexing service via a message queue. The indexing service processes tweets and updates the search index in a distributed search engine like Elasticsearch or Apache Solr.
- Sharded Indexing: Partition the search index by time (e.g., daily indices) or hashtags to distribute the load across multiple nodes. Older indices can be stored on slower storage systems to save costs while keeping recent indices on faster nodes.
- Query Optimization: Use inverted indexing to allow efficient keyword and hashtag search. Employ a ranking algorithm (e.g., BM25 or ML-based) to surface the most relevant tweets based on user engagement, recency, or other factors.
- Search Cache: Cache popular search queries and their results to reduce the load on the search engine.

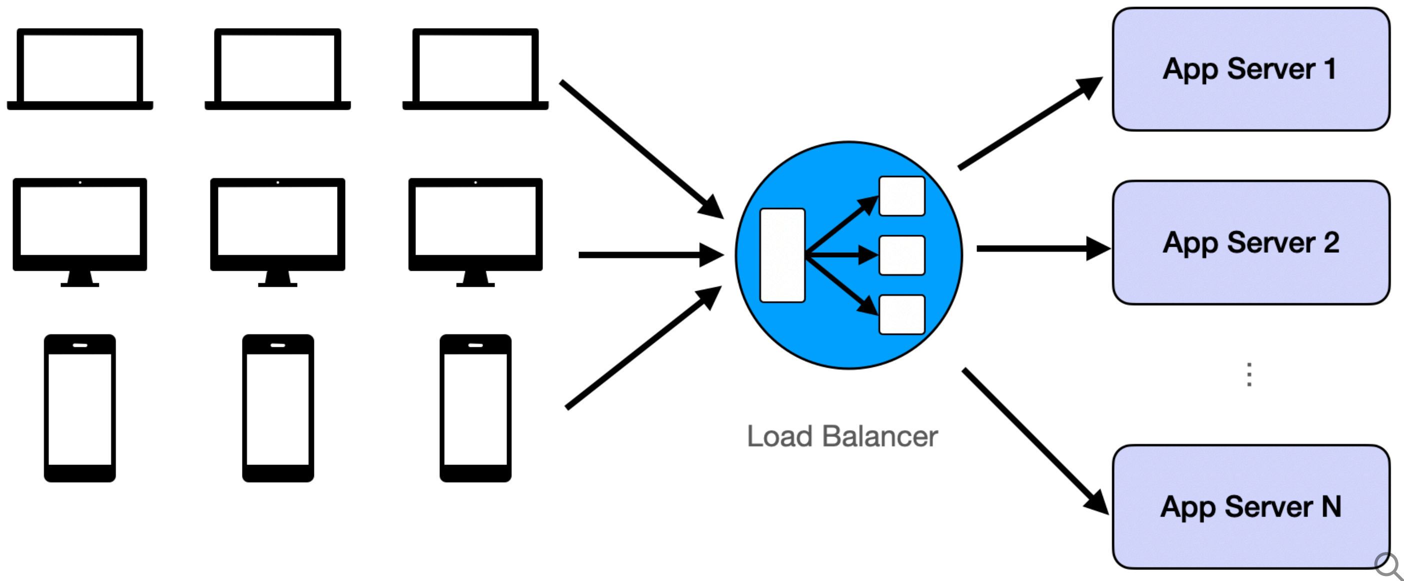
Core Design Challenges

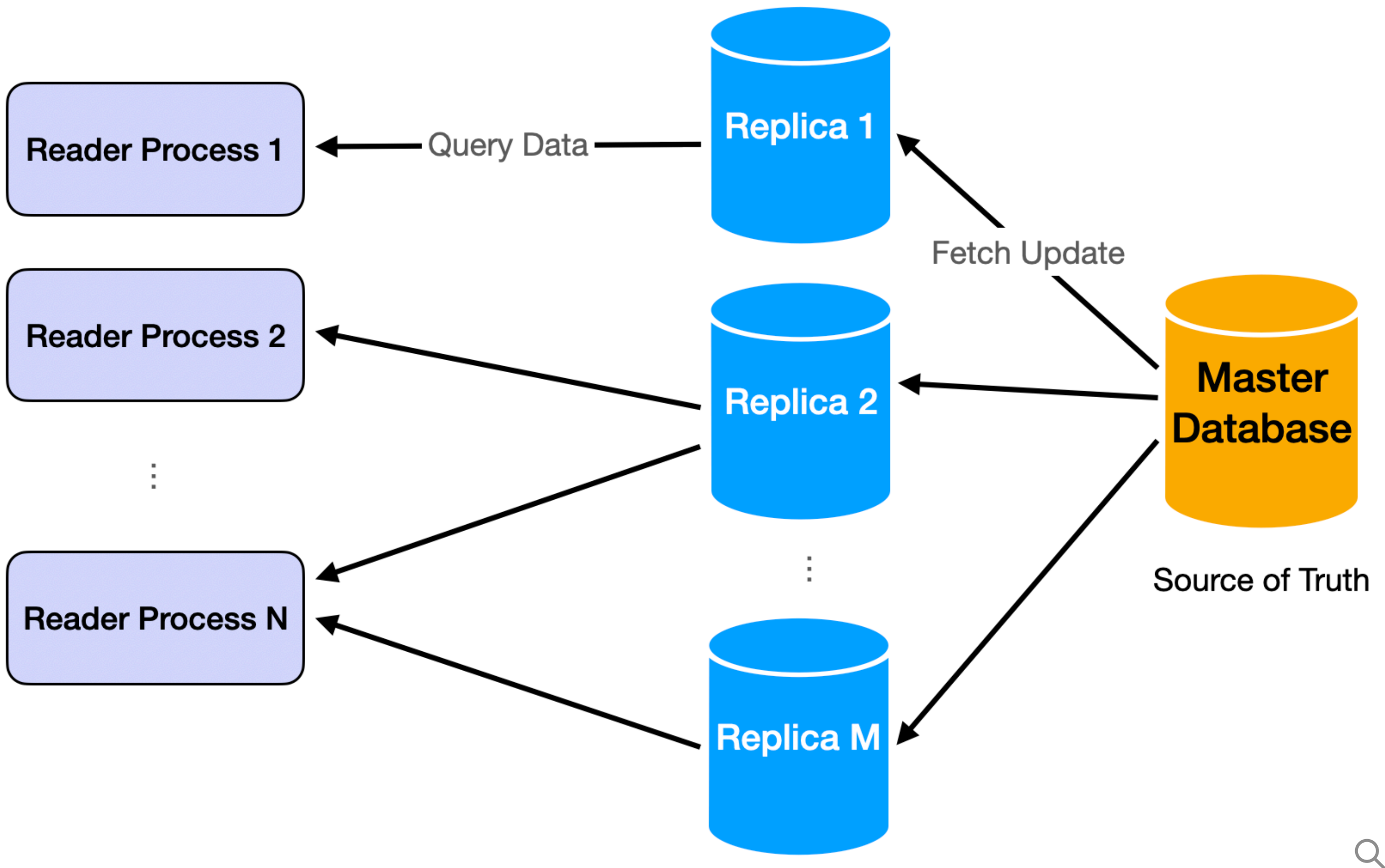
Now that we have a fair understanding of how to solve a system design problem, we should take a look at some of the most common system design problems. Like how our solutions are assembled from reusable building blocks, these challenges have several repeatable patterns.

Challenge 1: Too Many Concurrent Users

While a large user-base introduces many problems, the most common and intuitive one is that a single machine/database has a RPS/QPS limit. In all single-server demo apps you would see in a web dev tutorial, the server's performance will degenerate fast once the limit is exceeded.

The solution is also intuitive: repetition. We just repeat the same assets of our app and assign the users randomly to each replication. When the replicated assets are server logic, it's called load balancing. When the replicated asserts are data, it's usually called database replicas.

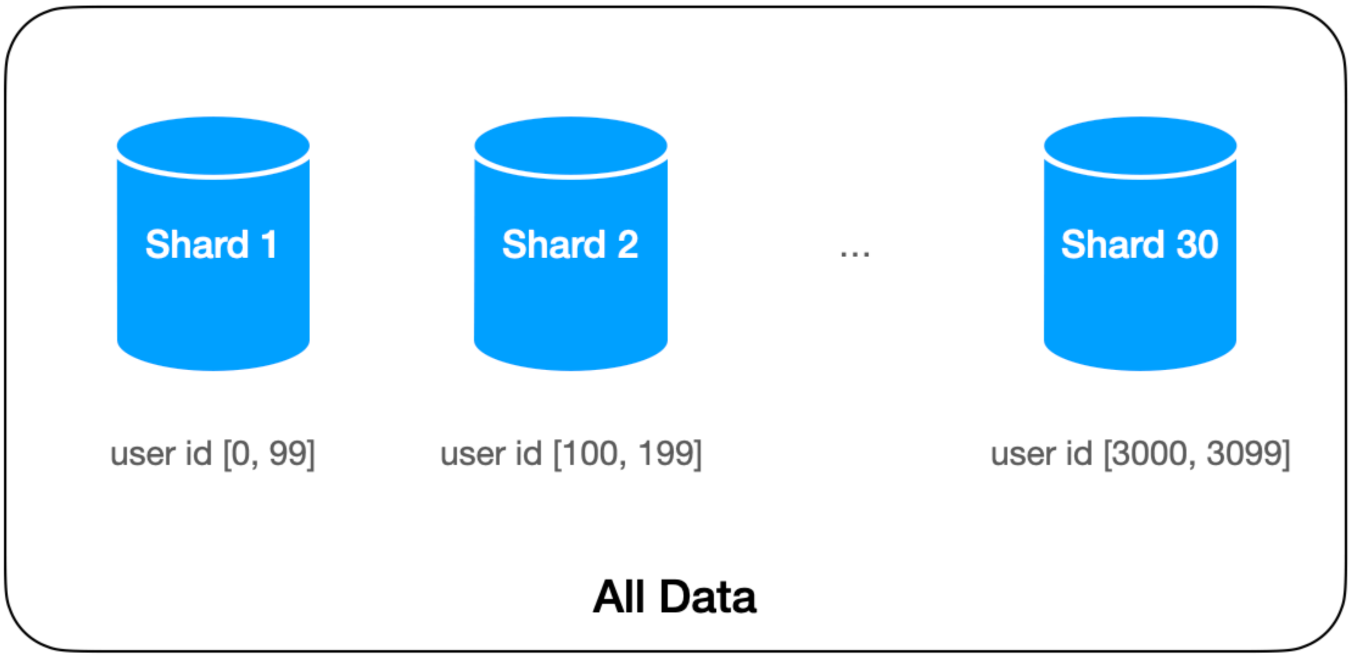




Challenge 2: Too Much Data to Move Around

The twin challenge of too many users is the issue of too much data. The data becomes 'big' when it's no longer possible to hold everything on one machine. Some common examples: Google index, all the tweets posted on Twitter, all movies on Netflix.

The solution is called sharding: partitioning the data by some logic. The sharding logic groups some data together, for instance, if we shard by user_id in Twitter, then all tweets from one user will be stored in the same machine.



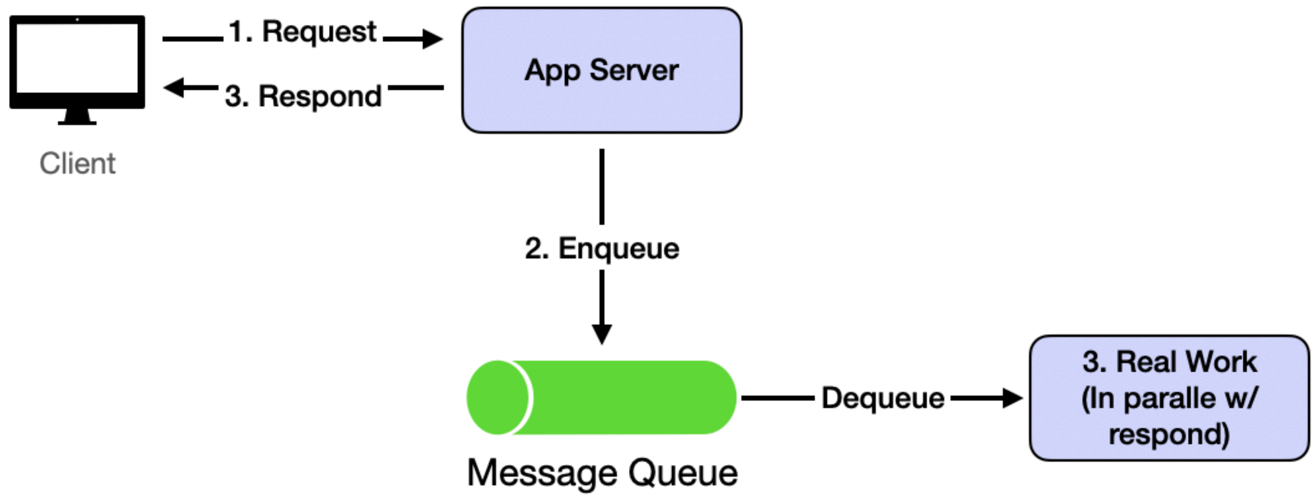
Challenge 3: The System Should be Fast and Responsive

Most user-facing apps must be quick. The response time should be less than 500ms. If it goes longer than 1 second, the user will have a poor experience.

Reading is usually fast after we have replication. Read requests are usually implemented as a query to an in-memory key-value dictionary beside HTTP protocols. Therefore, for many simple apps, the latency is mostly network round time.

Writing is where the challenge lies. Because most typical writing processes involve many data queries and updates, they last far longer than the 1-second limit. The solution is asynchrony: the write request is returned immediately after our server receives its data and puts the data in a queue. In the meantime, the actual processing continues in the back end. After receiving the response from the server, the client-side logic has the wiggle room for a speedy user experience. For example, it can show some UI before redirecting the user to read the result. This will usually take 1~2 seconds and is enough for the backend processing of the actual write request.

This is implemented by a message queue like Kafka.

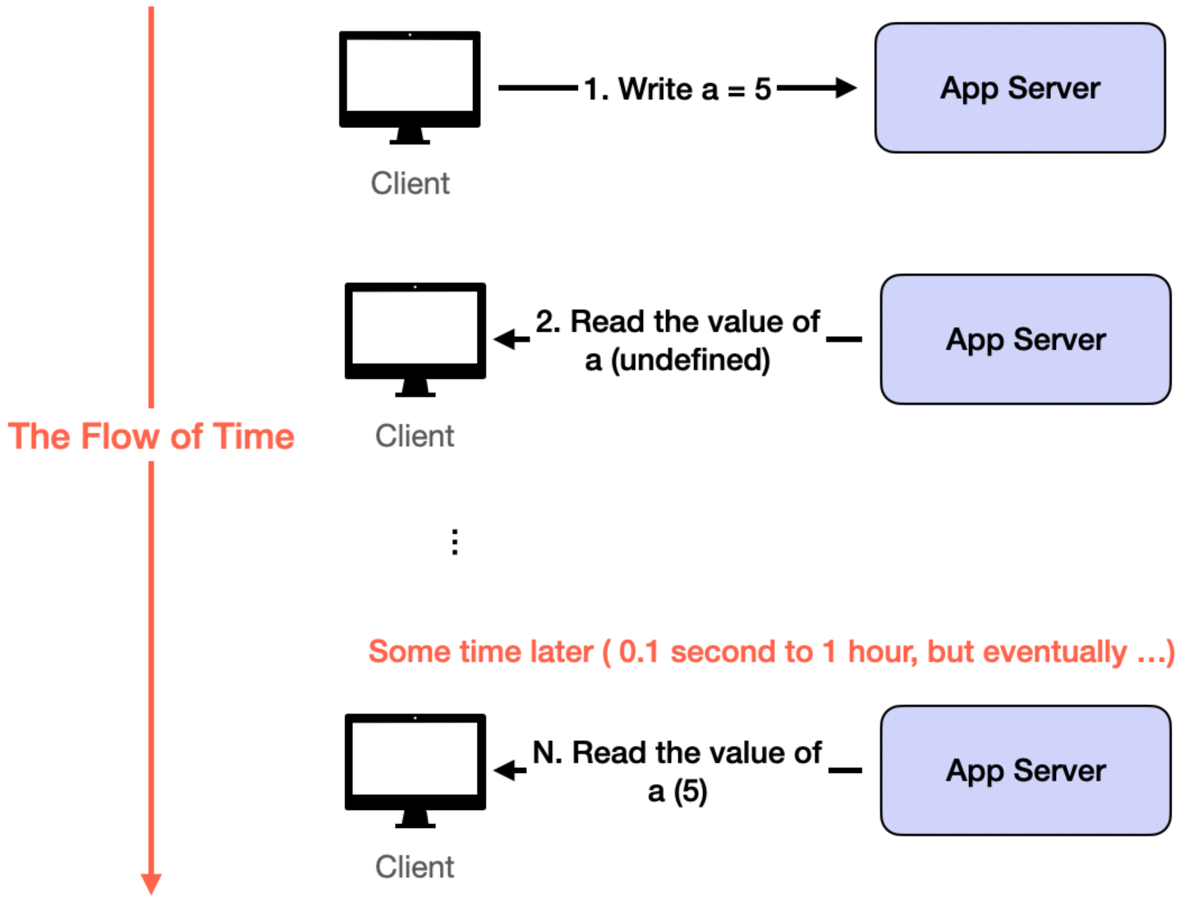


Challenge 4: Inconsistent (outdated) States

This challenge is a result of solving Challenge 1 and Challenge 2. With data replication and asynchronous data update, the read requests can easily see inconsistent data. Inconsistency usually means outdated: the user won’t see any random wrong data, but old versions or deleted data.

The solution is more on the application level than on the system level. Because the outdated read resulted from replication and asynchronous updates will eventually disappear when the servers catch up, we build the user experience so that seeing outdated data for a short period of time is OK. This is called eventual consistency.

Most Apps tolerate eventual consistency well. Especially compared with the alternatives: Losing data forever or being very slow. The exceptions are banking or payment-related apps. Any inconsistency is unacceptable, so the apps must wait for all processing to finish before returning anything to the users. That’s why such apps feel much slower than, say, Google Search.



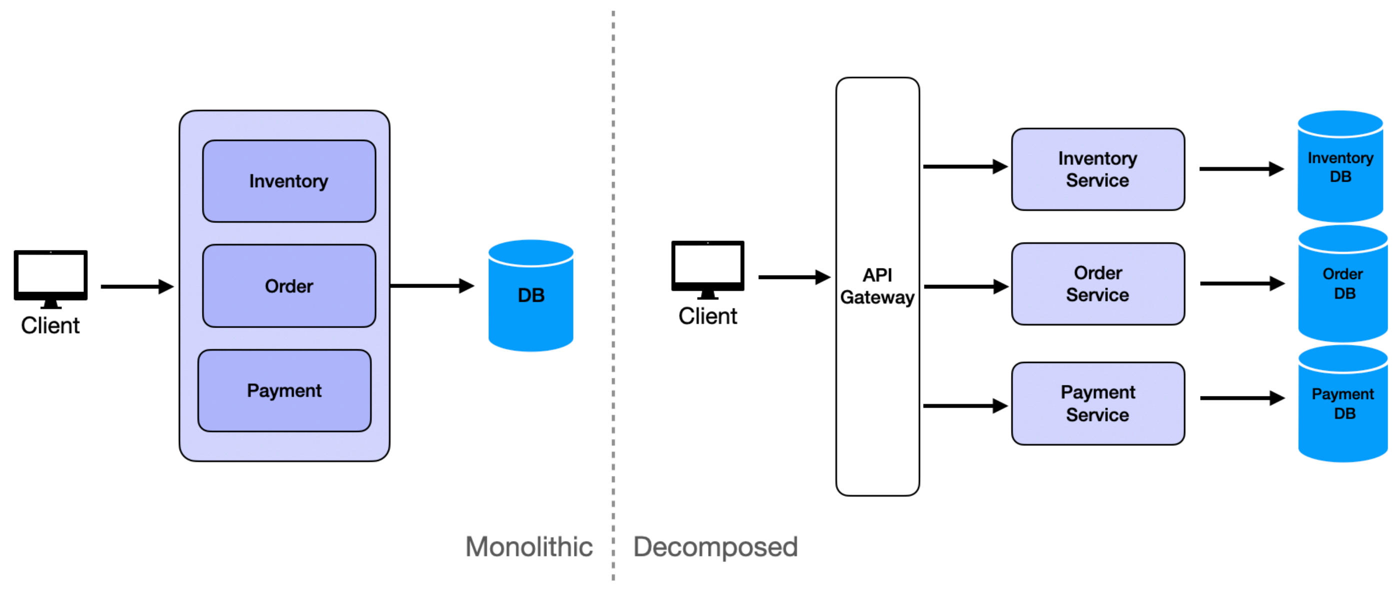
The Flow of Time
Client
1. Write a = 5
App Server
Client
2. Read the value of a (undefined)
App Server
⋮
Some time later (0.1 second to 1 hour, but eventually ...)
Client
N. Read the value of a (5)
App Server

Designing for Scale

Scaling a system effectively is one of the most critical aspects of satisfying non-functional requirements in system design. Scalability, in particular, is often a top priority. Below, we explore various strategies to achieve scalable system architecture.

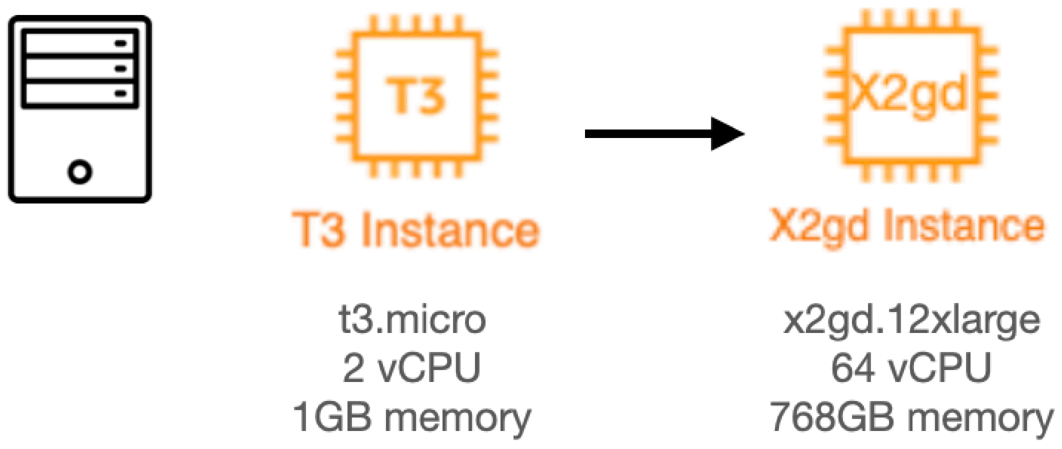
Decomposition

Decomposition involves breaking down requirements into microservices. The key principle is to divide the system into smaller, independent services based on specific business capabilities or requirements. Each microservice should focus on a single responsibility to enhance scalability and maintainability.



Vertical Scaling

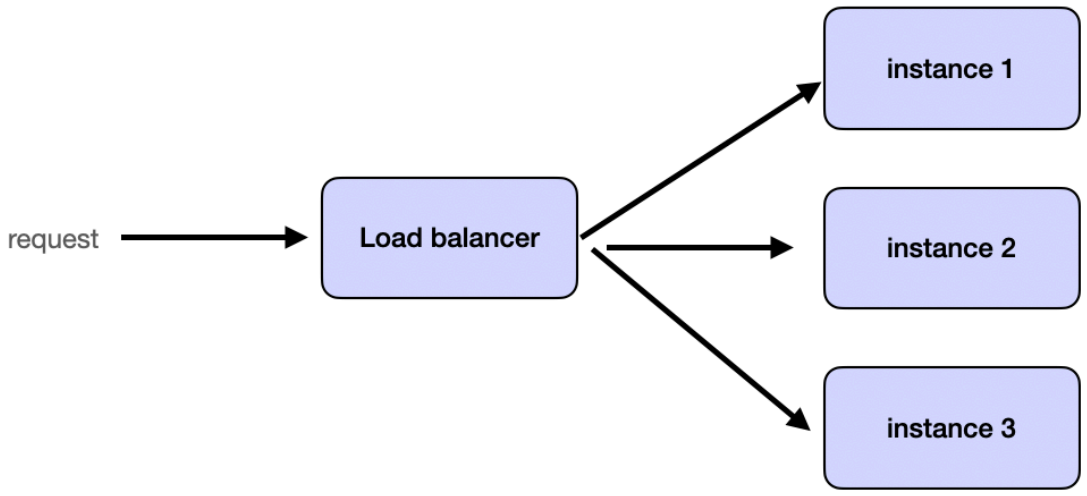
Vertical scaling represents the brute force approach to scaling. The concept is straightforward: scale up by using more powerful machines. Thanks to advancements in cloud computing, this approach has become much more feasible. While in the past, organizations had to wait for new machines to be built and shipped, today they can spin up new instances in seconds.



Modern cloud providers offer impressive vertical scaling options. For instance, AWS provides "Amazon EC2 High Memory" instances with up to 24 TB of memory, while Google Cloud offers "Tau T2D" instances specifically optimized for compute-intensive workloads.

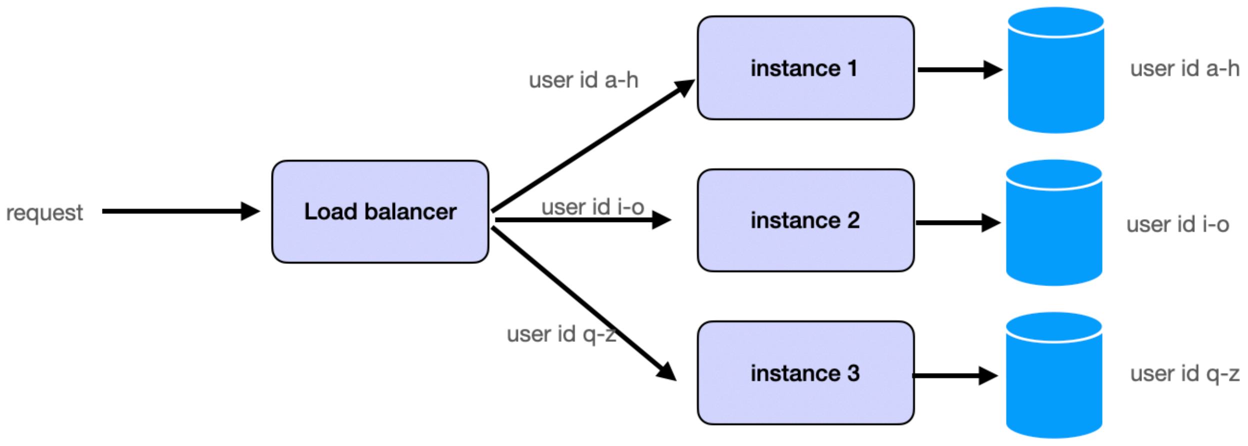
Horizontal Scaling

Horizontal scaling focuses on scaling out by running **multiple identical instances** of stateless services. The stateless nature of these services enables seamless distribution of requests across instances using load balancers.



Partitioning

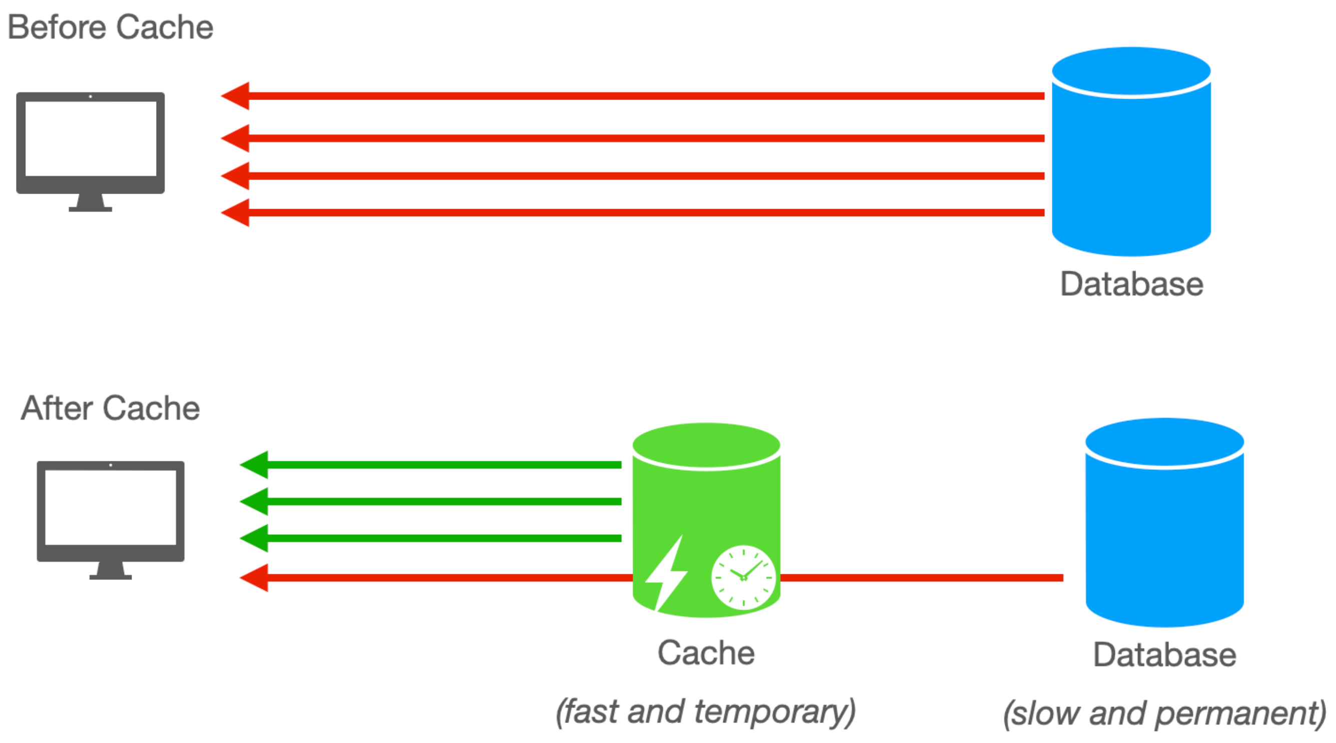
Partitioning involves **splitting requests and data into shards** and distributing them across services or databases. This can be accomplished by partitioning data based on user ID, geographical location, or another logical key. Many systems implement consistent hashing to ensure balanced partitioning.



We will cover partitioning in great detail in the Database Partitioning section.

Caching

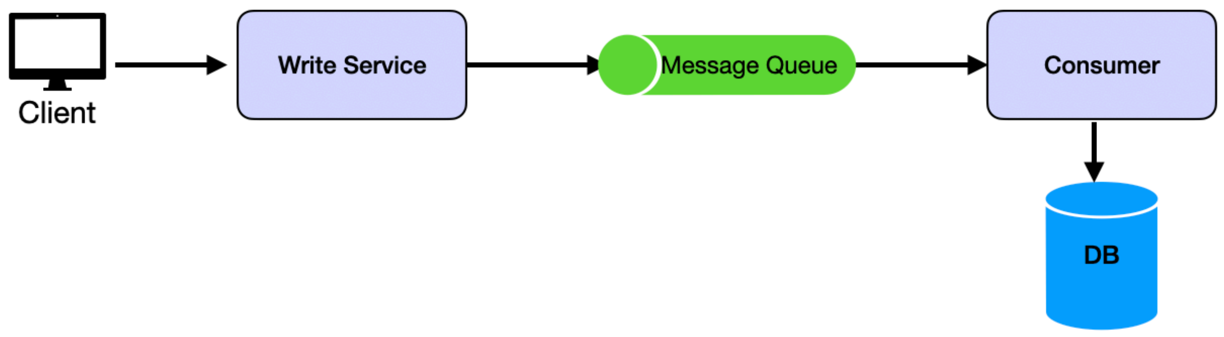
Caching serves to improve query read performance by storing frequently accessed data in faster memory storage, such as in-memory caches. Popular tools like Redis or Memcached can effectively store hot data to reduce database load.



We will cover caching in great detail in the Caching section.

Buffer with Message Queues

High-concurrency scenarios often encounter write-intensive operations. Frequent database writes can overload the system due to disk I/O bottlenecks. Message queues can buffer write requests, changing synchronous operations into asynchronous ones, thereby limiting database write requests to manageable levels and preventing system crashes.

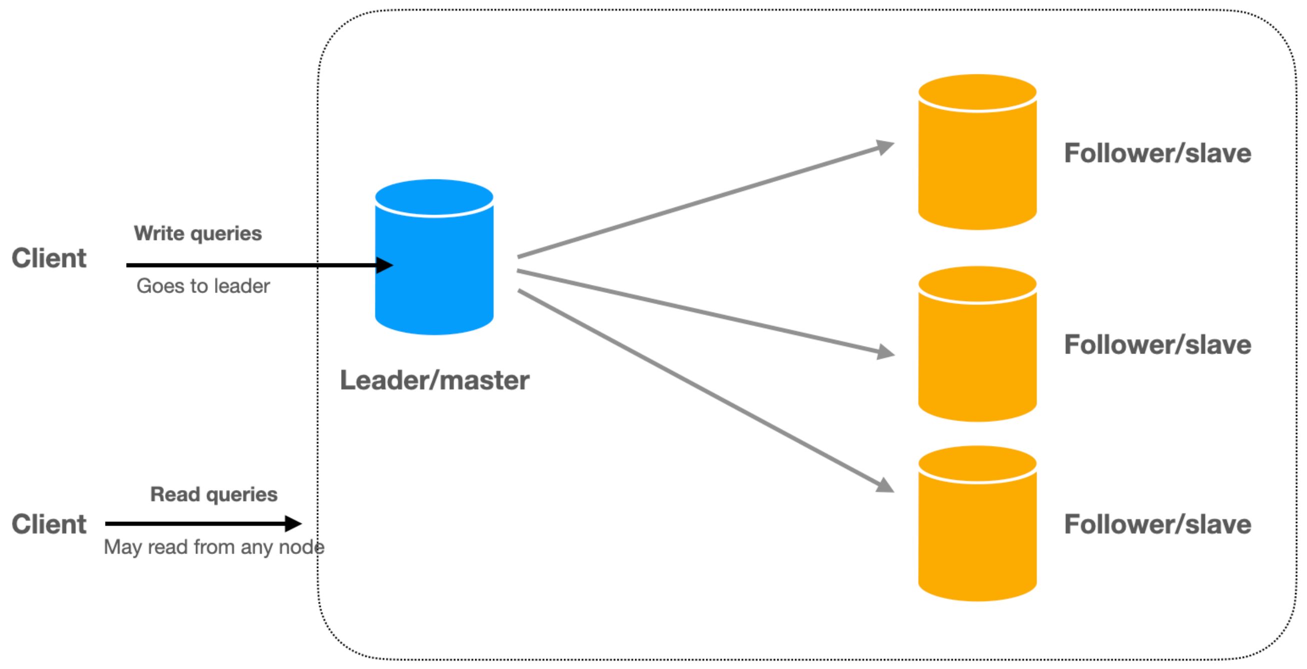


We will cover message queues in great detail in the Message Queues section.

Separating Read and Write

A system is read-heavy or write-heavy depends on the business requirements. For example, a social media platform is read-heavy because users read more than they write. On the other hand, an IOT system is write-heavy because users write more than they read. This is why we want to separate read and write operations to treat them differently.

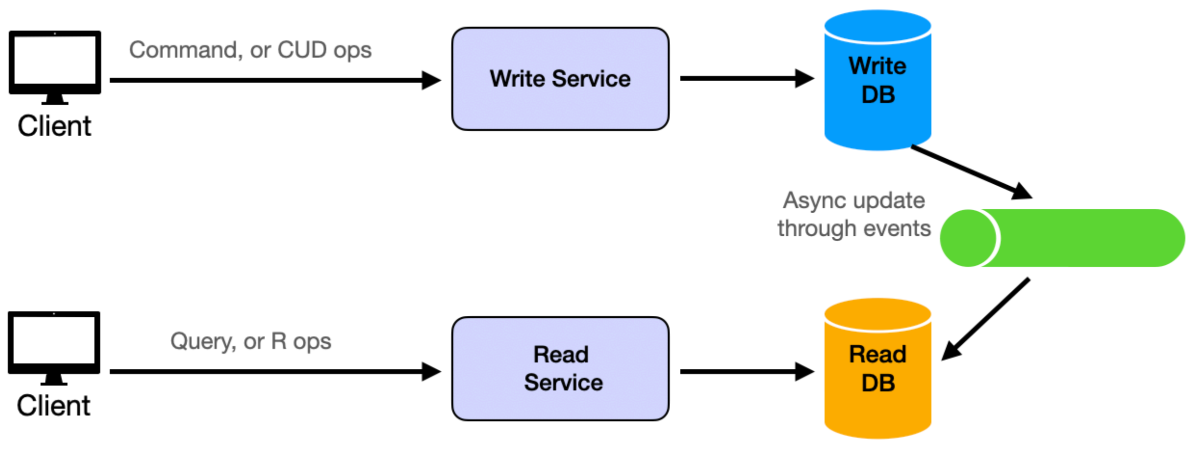
Read and write separation typically involves two main strategies. First, replication implements a **leader-follower architecture** where writes occur on the leader, and **followers provide read replicas**.



Second, the so-called **CQRS (Command Query Responsibility Segregation) pattern** takes read-write separation further by using completely different models for reading and writing data. In CQRS, the system is split into two parts:

- **Command Side (write side)**: Handles all write operations (create, update, delete) using a data model optimized for writes
- **Query Side (read side)**: Handles all read operations using a denormalized data model optimized for reads

Changes from the command side are **asynchronously propagated to the query side**.



CRUD = Create, Read, Update, Delete

For example, a system might use MySQL as the source-of-truth database while employing Elasticsearch for full-text search or analytical queries, and asynchronously sync changes from MySQL to Elasticsearch using MySQL binlog **Change Data Capture (CDC)**.

Combining Techniques

Effective scaling usually requires a multi-faceted approach combining several techniques. This starts with decomposition to break down monolithic services for independent scaling. Then, partitioning and caching work together to distribute load efficiently while enhancing performance. Read/write separation ensures fast reads and reliable writes through leader-replica setups. Finally, business logic adjustments help design strategies that mitigate operational bottlenecks without compromising user experience.

Adapting to Changing Business Requirements

Adapting business requirements offers a practical way to handle large traffic loads. While not strictly a technical approach, understanding these strategies demonstrates valuable experience and critical thinking skills in an interview setting.

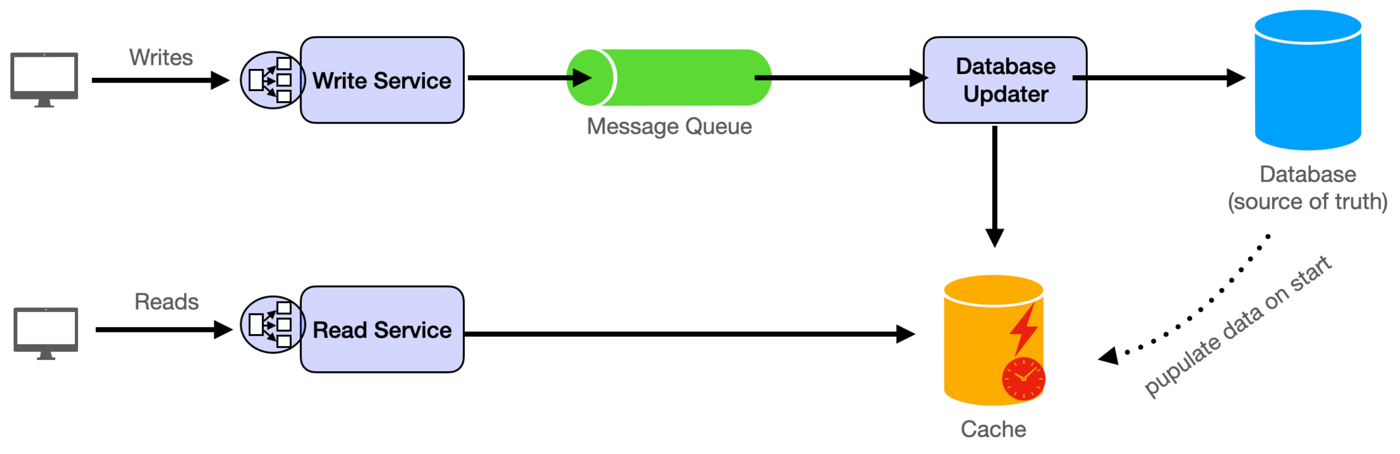
Consider a weekly sales event scenario: Instead of running all sales simultaneously for all users, the load can be distributed by allocating specific days for different product categories for specific regions. For instance, baby products might be featured on Day 1, followed by electronics on Day 2. This approach ensures more predictable traffic patterns and enables better resource allocation such as pre-loading the cache for the upcoming day and scaling out the read replicas for the specific regions.

Another example involves handling consistency challenges during high-stakes events like eBay auctions. By temporarily displaying bid success messages on the frontend, the system can provide a seamless user experience while the backend resolves consistency issues asynchronously. Users will eventually see the correct status of their bid after the auction ends.

While these are not technical solutions, bringing them up in an interview demonstrates your ability to think through the problem and provide practical solutions.

Master Template

Here is the common template to design scalable services (and therefore solve many system design problems):



A very high-level takeaway:

- Write to message queue and have the consumers/workers update the database and cache
- Read from cache

Let's take a look at it step-by-step.

Component Breakdown

Stateless Services

- Scalable and stateless, these services can be expanded by adding new machines and integrating them through a load balancer.
- **Write Service**:
  - Receives client requests and forwards them to the message queue.
- **Read Service**:
  - Handles read requests from clients by accessing the cache.

Database

- Serves as cold storage and the source of truth. However, we do not normally read directly from the database since it can be quite slow when the request volume is high.

Message Queue

- A buffer between writer services and data storage.
- **Producers**:
  - Comprised of write services that send data changes to the queue.

- **Consumers**:
  - Involved in updating both the database and the cache.
  - **Database Updater**:
    - Asynchronous workers update the database by retrieving jobs from the message queue.
  - **Cache Updater**:
    - Asynchronous workers refresh the cache by fetching jobs from the message queue.

Cache

- Facilitates fast and efficient read operations.

Now let's take a look in detail.

Dataflow path

Almost all applications can be broken down into read requests and write requests.

Because read and write have completely different implications (read doesn’t mutate; write mutates database), we discuss write path and read path separately.

Read path

For modern large-scale applications with millions of daily users, we almost always read from cache instead of from the database directly. The database acts as a permanent storage solution. Asynchronous jobs frequently transfer data from the database to the cache.

Write path

Write requests are pushed into a /fundamentals/message-queue, allowing backend workers to manage the writing process. This approach balances the processing speeds of different system components, offering a responsive user experience.

Message queue

Message queue is essential to scaling out our system to handle write requests.

- Producers: Insert messages into the queue.
- Consumers: Retrieve and process messages asynchronously.

The necessity of message queues arises from:

- Varying Processing Rates: Producers and consumers handle data at different speeds, necessitating a buffer.
  - e.g., the frontend posts comments a lot faster than the backend can write to the db
  - e.g., the frontend booking request is a lot faster than the backend booking service (needs to contact 3rd party)
- Fault Tolerance: They ensure the persistence of messages, preventing data loss during failures.
  - imagine having the write service calling the db updater (workers) directly. If the request fails, the request is lost
  - with mq, messages are persisted in the queue so if workers are down