How to design a cold storage tier for rarely accessed data?....377
What is read repair in eventually consistent databases?....378
How to handle tombstone records in databases like Cassandra?....379
How to implement secondary indexes in distributed databases?....380
Caching & Queues....381
What caching strategies do you know?....381
How does write-through caching differ from write-back?....383
What is cache invalidation and why is it hard?....384
How to prevent cache stampedes?....385
How would you design a distributed message queue?...386
Kafka vs RabbitMQ: When to use which?....387
Kafka vs RabbitMQ....387
How do you implement a rate limiter using Redis?....388
How would you implement delayed message delivery?....389
How to design for exactly-once processing in queues?...390
How to implement a multi-layer cache hierarchy?....391
Compare in-memory vs distributed caching?....392
What's the difference between fanout-on-write and fanout-on-read in queues?....393
How to design dead-letter queues and retry queues?....394
How to handle message ordering guarantees in distributed queues?....395
How to implement FIFO queues at scale?....396
Scalability & Reliability....397
How do you design for peak traffic vs average traffic?...397

How would you handle sudden traffic spikes?.....399
How to detect and recover from cascading failures?.....400
How do you scale a read-heavy database?.....401
How to design for zero downtime deployments?.....402
How to implement auto-scaling in Kubernetes?.....403
What is chaos engineering and why is it useful?.....405
How to design a load-shedding mechanism?.....406
How to handle brownout and blackout conditions in services?.....407
How to test scalability before launch?.....408
How to design a service that can scale from 1K to 1B users?.....409
How to handle the thundering herd problem?.....411
APIs & Interfaces.....412
How do you design a REST API for a messaging service?.....412
REST vs GraphQL: Which would you choose?.....414
How would you handle API versioning?.....415
How to design a pagination system?.....416
How to implement retries with exponential backoff?.....417
How to secure public APIs against abuse?.....418
How to design an API for partial responses?.....419
How would you implement ETag-based caching for APIs?.....420
What is HATEOAS and when would you use it?.....421
How to implement batch API requests for efficiency?....422
How to design idempotent APIs?.....424
Security.....425
How would you secure an API using OAuth2?.....425
How to encrypt data at rest and in transit?.....426

How to design for Zero Trust security?.....427
How to prevent replay attacks?.....428
How to secure a multi-tenant SaaS platform?.....429
How to design secure secret management?.....430
How to handle API key rotation at scale?.....431
What's the difference between symmetric and asymmetric encryption in practice?.....432
How would you implement mutual TLS (mTLS)?.....433
How to design a WAF for a large-scale API?.....434
How to defend against SQL injection and XSS?.....436
How to implement signed URLs for temporary access?.437
How to design rate limiting for security purposes?.....438
Operations & Monitoring……...439
How do you design a centralized logging system?.....439
How to implement distributed tracing?.....440
How to set up automated health checks for microservices?.....441
How to design an alerting system to reduce noise?.....442
How to monitor SLIs for a high-traffic service?.....443
How to design a metrics aggregation pipeline?.....444
How to perform root cause analysis in distributed outages?.....445
How to design a service for observability from day one?.....446
How to handle metric cardinality issues in Prometheus?.....447
How to design custom application-level monitoring?....448
Deployment & Infrastructure Automation…….449
How do you design a CI/CD pipeline for microservices?.....449
Blue/Green vs Canary deployments…….450

How to perform safe database schema changes during deployments?.....451
How to implement feature flags safely in production?...452
How to design infrastructure as code using Terraform?.453
How to handle rollbacks in production deployments?...454
How to design for immutable infrastructure?.....455
How to orchestrate deployments across multiple regions?.....456
How to do traffic shadowing before full rollout?.....457
Tradeoffs & Cost Awareness……..458
How to balance cost, performance, and user experience in architecture…….459
When would you choose eventual consistency over strong consistency?.....460
How to optimize cloud costs without degrading performance?.....461
How to tradeoff between caching and database reads…462
How to decide between managed services and self-hosted solutions……..463
How to choose between monolith and microservices for a given project?.....464
How to plan infrastructure costs for unpredictable workloads?.....465

What is System Design?

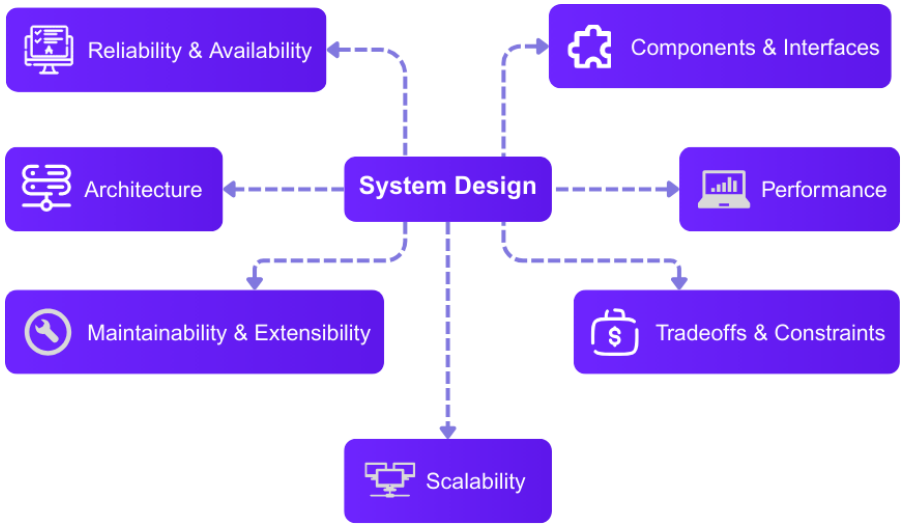
System design is the process of **defining the architecture, components, and interactions of a system** to satisfy specified requirements while meeting goals such as scalability, reliability, performance, and maintainability. It's the blueprint that guides how software or a system works end-to-end.

In simpler terms:

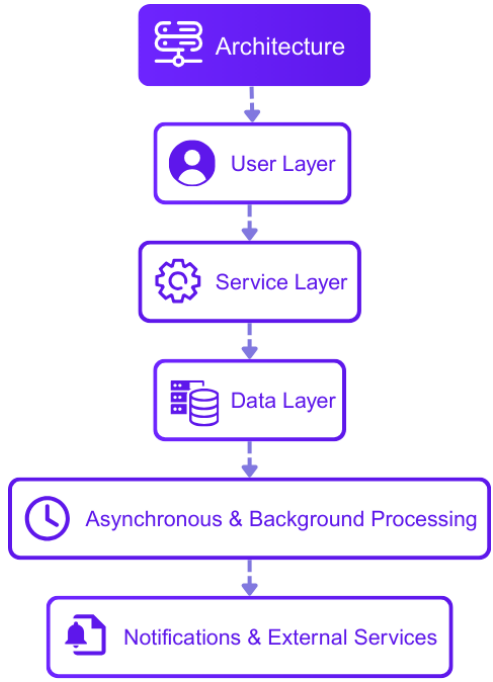
System design is about answering the question:

"How should this system be built so it can handle real-world challenges at scale?"

Key Aspects of System Design



Architecture

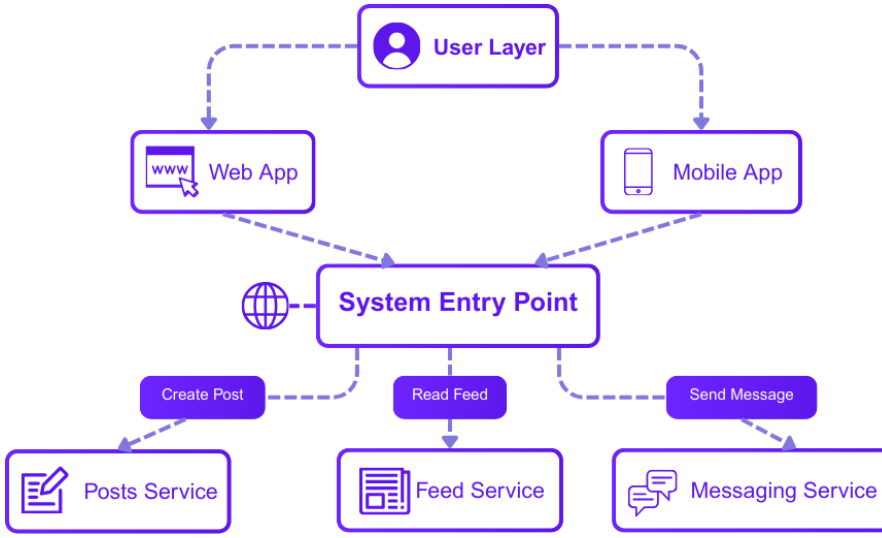


Architecture is the big-picture layout of a system. It shows how all the pieces fit together and communicate with each other.

Think of it like the blueprint of a house:

- Rooms, plumbing, and wiring need to be in the right place for everything to work. In software, good architecture ensures the system runs smoothly and can grow over time.

User Layer



The user interacts with the system through a client app, either web or mobile. The system receives requests like creating posts, reading feeds, or sending messages, and directs them to the appropriate services.

Users can perform actions like:

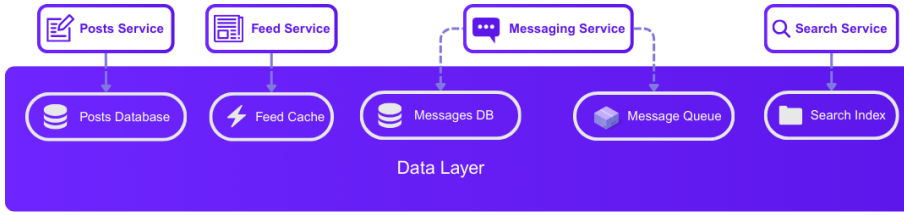
- Posting new content → goes to the **Posts Service**
- Viewing their timeline → goes to the **Feed Service**
- Sending messages → goes to the **Messaging Service**

Service Layer

This layer contains the **main services** that handle specific responsibilities:

- **Posts Service:** Creates and stores posts in a database.
- **Feed Service:** Aggregates posts from friends and delivers timelines. Often uses a cache to speed up repeated requests.
- **Messaging Service:** Handles sending and receiving private messages. Uses a message queue to ensure reliable delivery.
- **Notifications Service:** Sends alerts about likes, comments, and mentions.
- **Search Service:** Enables searching posts, users, or hashtags efficiently using a search index.

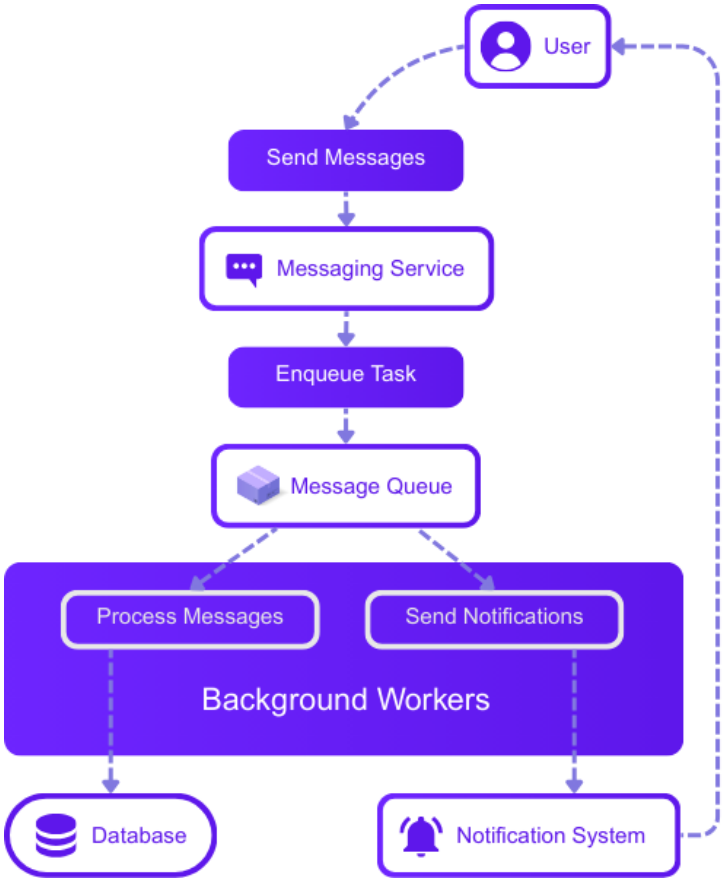
Data Layer



Services use different data storage options based on their needs:

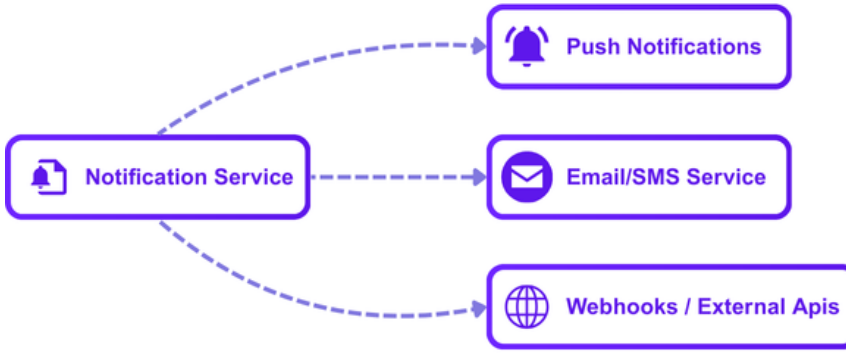
- **Databases:** Persist important data like posts or messages.
- **Caches:** Store frequently accessed data for faster responses, e.g., feed cache.
- **Search Index:** Optimized for search queries to quickly find posts, users, or hashtags.

Asynchronus & Background Processing



Some tasks, like sending messages or notifications, are handled asynchronously using queues. This improves reliability and system performance, ensuring users don't experience delays.

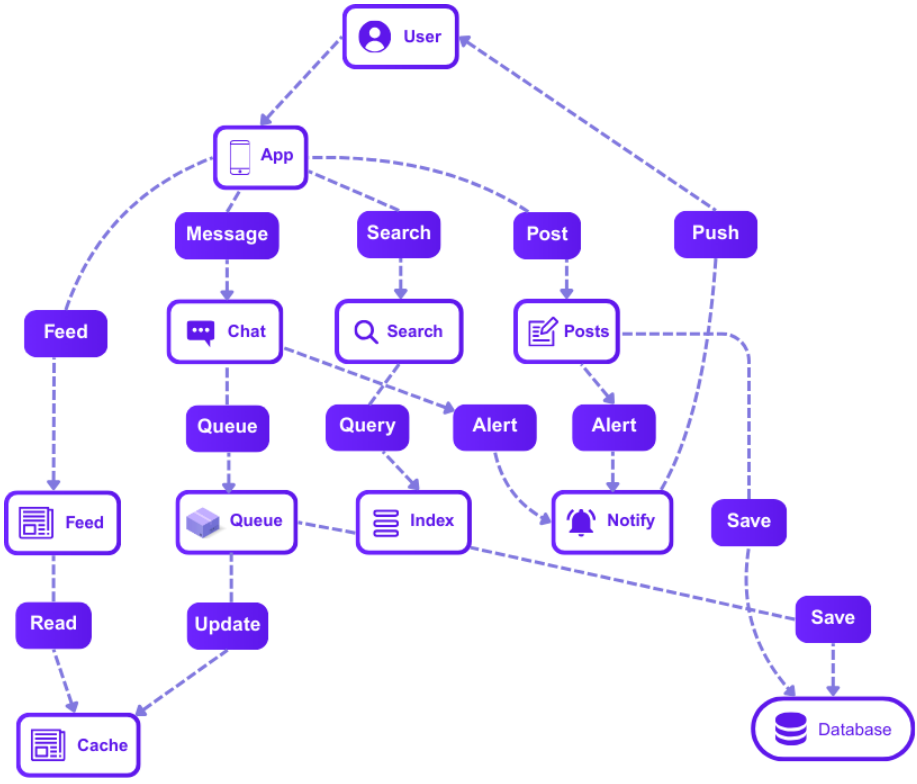
Notifications & External Services



External-facing services like push notifications deliver alerts in real time. They often sit at the edge of the system and interact with the core services.

Putting it together

The architecture ensures smooth communication between users, services, databases, caches, and external systems. Each service is independent but connected, making it easier to scale, maintain, and improve the system over time.



Components & Interfaces

Components are the building blocks of a system. Breaking a system into smaller, manageable parts makes it easier to design, maintain, and scale. Interfaces define how these components communicate with each other, like roads connecting different parts of a city. Together, they make sure each part does its job while working seamlessly with the rest of the system.

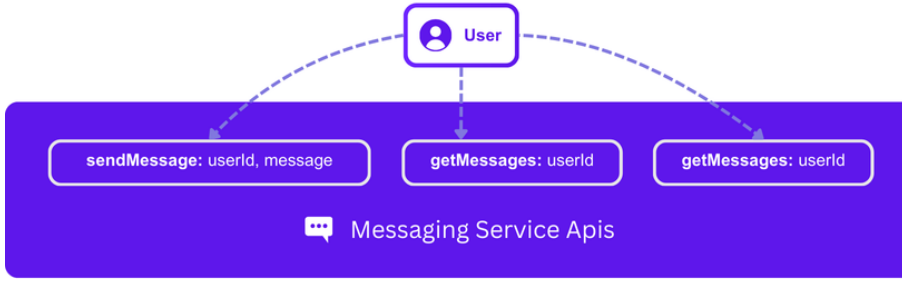
- Makes the system easier to understand and maintain
- Helps isolate failures so one part does not bring down the entire system
- Allows independent development of components by different teams
- Improves reusability because components can be used in multiple places

Example: Messaging Service

Suppose you are designing a messaging system.

Here is how components and interfaces might look:

APIs (Interfaces):



- `sendMessage(userId, message)` sends a message to a user
- `getMessages(userId)` retrieves the last N messages for a user
- `markAsRead(messageId)` marks a message as read

These APIs define how other services or clients interact with the messaging service.

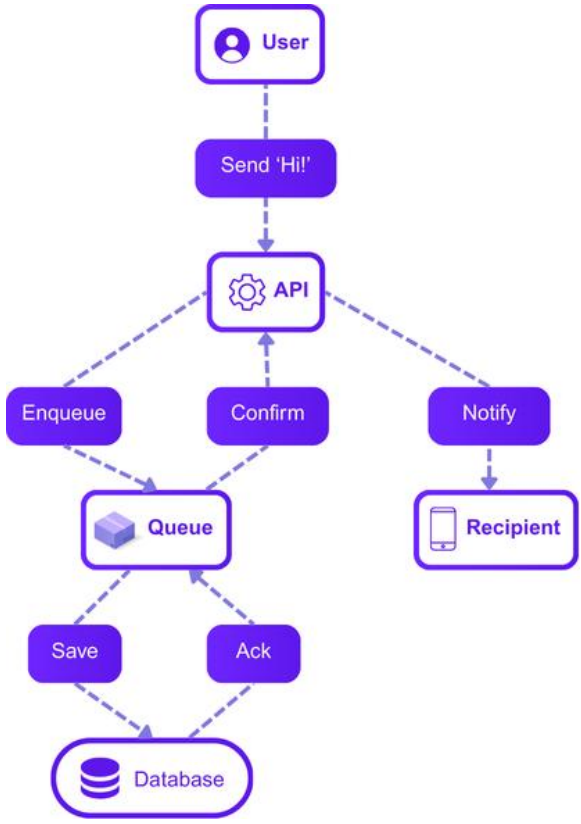
Components

- **Message Queue (e.g., Kafka)** ensures messages are delivered reliably and asynchronously
- **Database (e.g., PostgreSQL)** stores messages persistently
- **Notification System** sends alerts for unread messages to recipients
- **Service Layer** contains the logic for validating, storing, and retrieving messages

How They Work Together?

- A user sends a message using `sendMessage()`
- The message is placed in the **Message Queue** for reliable delivery
- The **Database** stores the message
- The **Notification System** alerts the recipient about the new message
- The recipient retrieves messages using `getMessages()`

This separation allows the system to continue working even if one part fails.



- Consider possible failure points. For example, what happens if the database crashes, the queue is full, or a notification fails
- Implement graceful error handling and retries to make the system resilient
- Keep APIs simple and consistent so other components or clients can use them easily
- Document each component and interface clearly to make scaling and onboarding easier
- Design for future extensibility, such as adding message reactions or read receipts without changing core logic

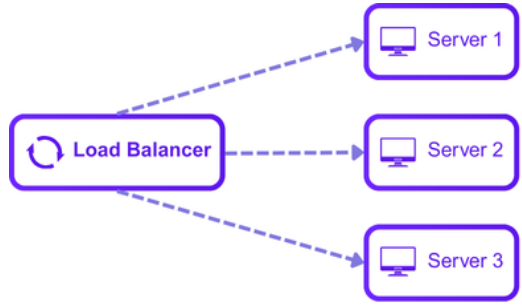
Scalability

Scalability is the ability of a system to handle growth in users, traffic, or data without breaking or slowing down. A scalable system performs well whether it has 100 users or 1 million. Designing for scalability ensures your system can grow over time without needing a complete rewrite.

- Prevents crashes or slowdowns as the number of users increases
- Reduces the need for frequent major redesigns
- Improves user experience by keeping response times fast
- Makes it easier to plan for future growth

Types of Scaling

Vertical Scaling:



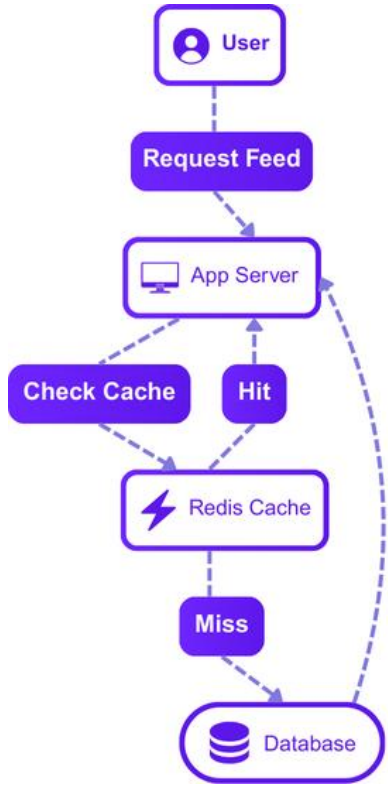
Vertical scaling, or scaling up, means upgrading your existing servers to be more powerful. For example, adding more CPU, RAM, or storage to a single server. This is simple to implement but has limits because there is a maximum capacity for a single machine.

Horizontal Scaling:

Horizontal scaling, or scaling out, means adding more servers to handle increased load. For example, running multiple instances of your web or database servers behind a load balancer. This approach is more flexible and allows the system to grow almost indefinitely, but it requires careful management of data consistency and distribution.

Scalability Techniques

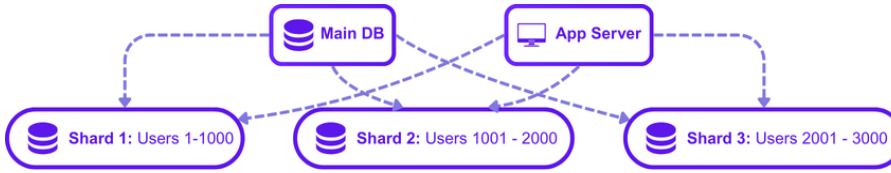
Caching:



Store frequently accessed data in memory for faster access.

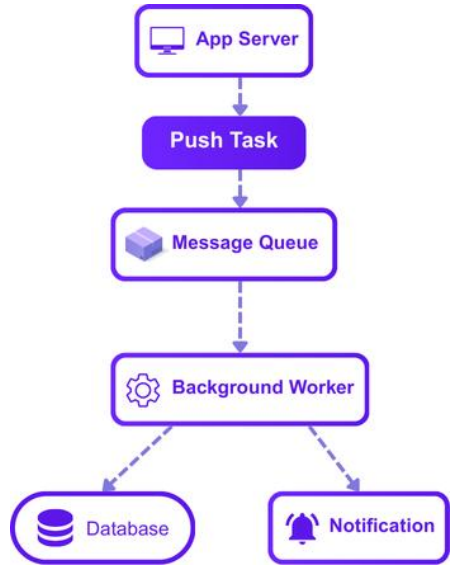
Example: Using Redis to cache user feeds or search results.

Database Sharding: Split a database into smaller, independent pieces (shards) to distribute load.



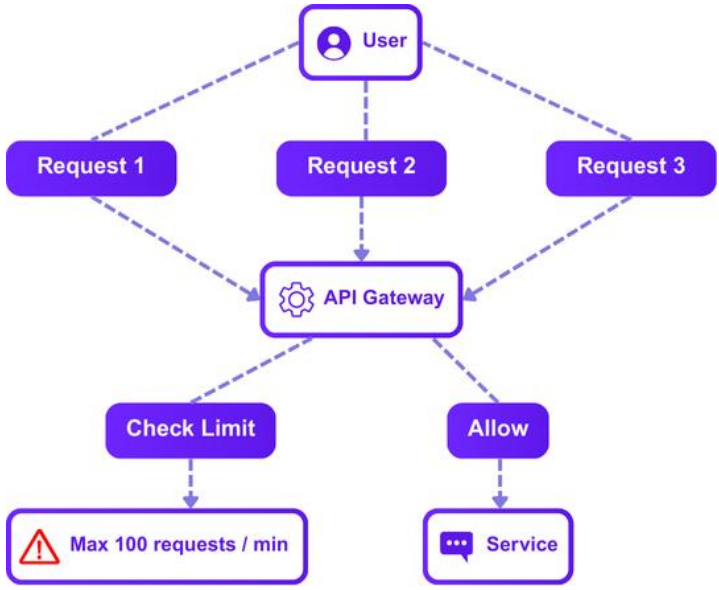
Example: Each shard stores messages for a range of user IDs.

Asynchronous Processing: Use queues and background workers to handle tasks that do not need to happen immediately.



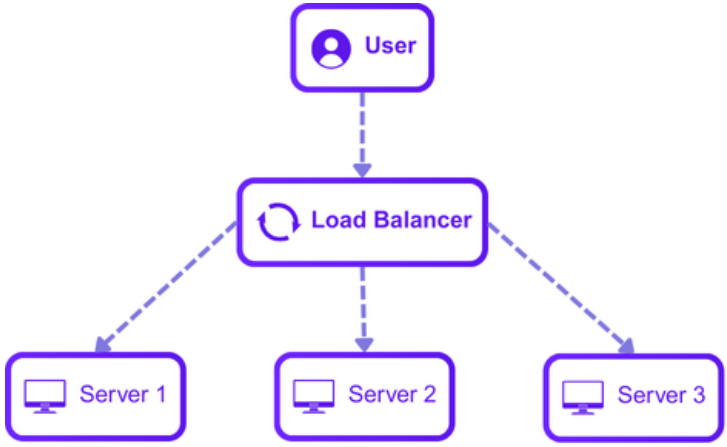
Example: Sending notifications after a message is saved in the database.

Rate Limiting: Control how many requests a user or service can make to prevent overloading the system.



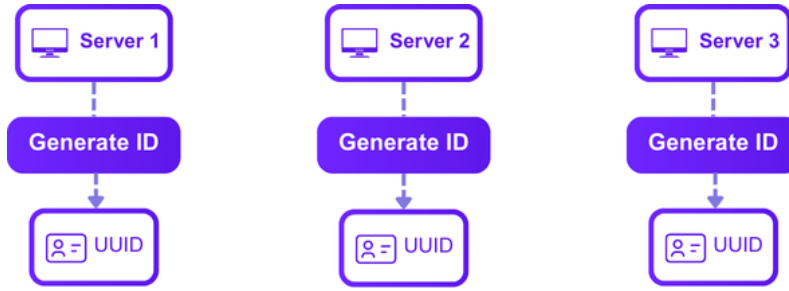
Example: Allowing each user to send 100 messages per minute.

Load Balancing: Distribute incoming requests across multiple servers to prevent any single server from being overwhelmed.



Example: Using Nginx or AWS ELB to distribute traffic across web servers.

Designing Unique ID Generators: Generate IDs in a way that avoids conflicts across multiple servers.



Example: Using UUIDs or Snowflake IDs for messages or posts.

Tips:

- **Start simple:** scale vertically first if your traffic is low, then scale horizontally as you grow.
- **Combine multiple techniques:** caching, asynchronous processing, and load balancing often work best together.
- Monitor your system to identify bottlenecks before they become critical.
- Always plan for **both current load and future growth**.

Reliability & Availability

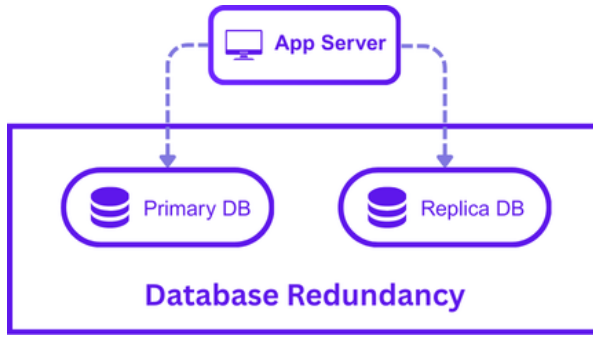
Reliability is the ability of a system to operate correctly and consistently over time. Availability is the measure of how often a system is accessible and functional for users. Together, they ensure that users can trust your system to work as expected, even when things go wrong.

- Prevents downtime that frustrates users or impacts business.
- Ensures critical data is not lost during failures.
- Builds trust and confidence in the system.

Reduces the impact of unexpected hardware or software failures.

Key Concepts

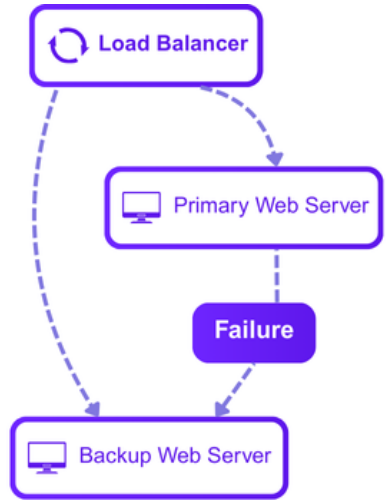
Redundancy:



Duplicate critical components so that if one fails, another can take over.

- **Example:** Two database servers in a primary/replica setup.

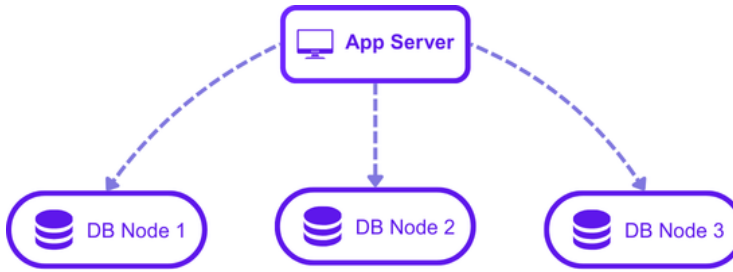
Failover:



Automatically switch to a backup component when the main one fails.

- **Example:** If the primary web server crashes, traffic is redirected to a standby server.

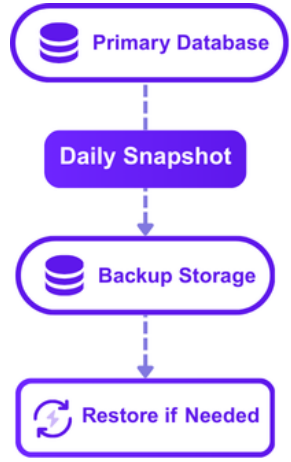
Replication:



Keep copies of data in multiple locations to prevent data loss and improve read performance.

- **Example:** Replicating user messages across multiple database nodes.

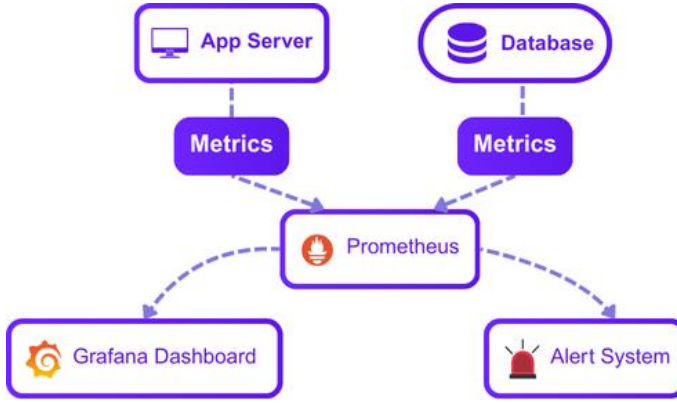
Backups:



Regularly store copies of data to recover from failures or corruption.

- **Example:** Daily snapshots of the posts database.

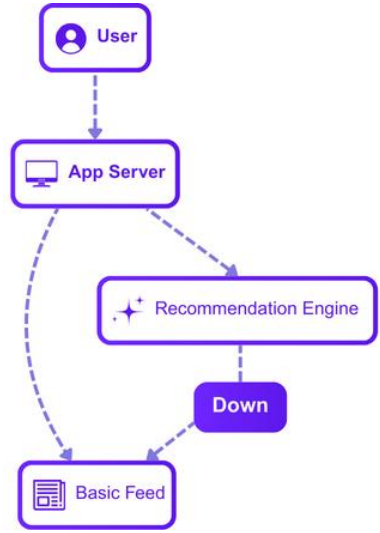
Health Checks & Monitoring:



Continuously monitor system components and automatically alert if something goes wrong.

- **Example:** Using Prometheus or Grafana to track server uptime and response times.

Graceful Degradation:



Ensure the system continues to function in a reduced capacity if some components fail.

- **Example:** If the recommendation engine is down, the feed still shows basic posts.

Practical Tips:

- Design for **no single point of failure**. Every critical component should have a backup or replica.
- Test failover mechanisms regularly to ensure they work when needed.
- Combine **replication, redundancy, and monitoring** for maximum reliability.
- Document recovery procedures so your team can respond quickly during incidents.
- Consider **user experience** during failures: inform users of limited functionality instead of crashing completely.

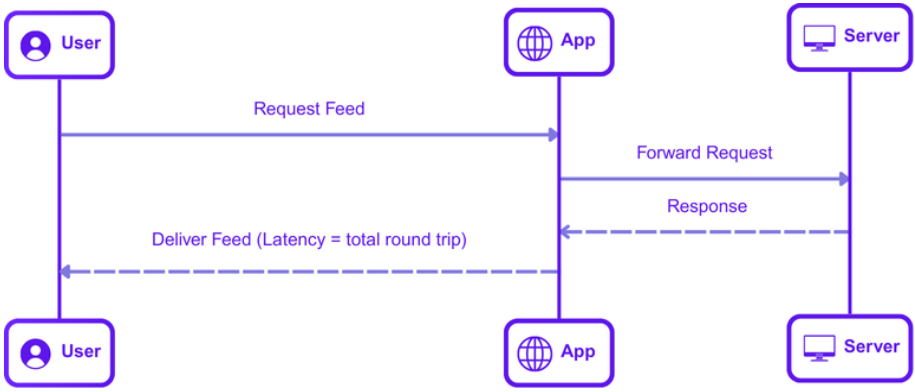
Performance:

Performance measures how fast and efficiently a system responds to requests and processes data. A high-performance system delivers results quickly, handles many requests at once, and provides a smooth experience for users.

- Users expect fast responses, especially in social media, messaging, or e-commerce apps.
- Poor performance can lead to frustration, lost engagement, or lost revenue.
- Efficient systems use resources wisely, reducing costs.
- Good performance supports scalability and reliability.

Key Concepts

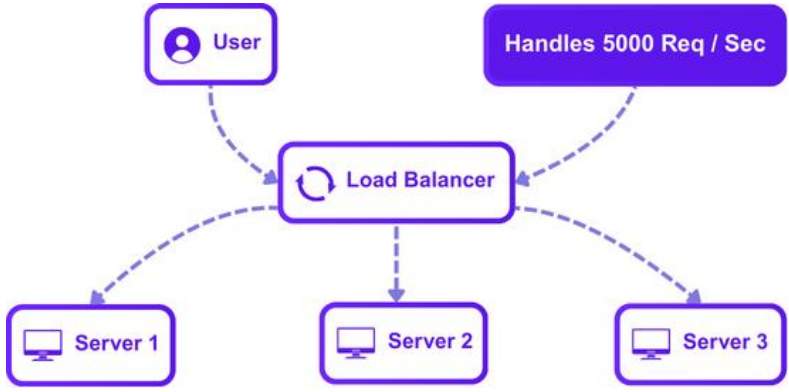
Latency:



The time it takes for a request to travel from the client to the server and back.

- **Example:** How long it takes to load a user's feed after opening the app.

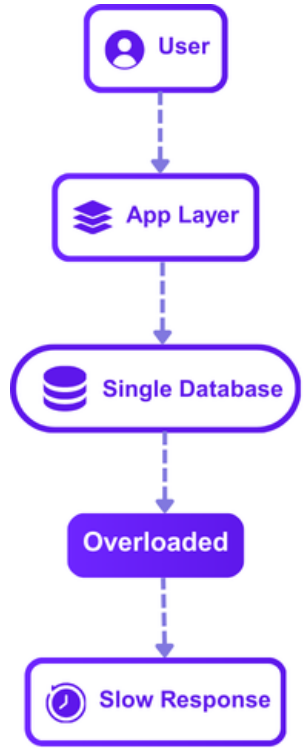
Throughput:



The number of requests a system can handle in a given period.

- **Example:** A server can process 5000 messages per second.

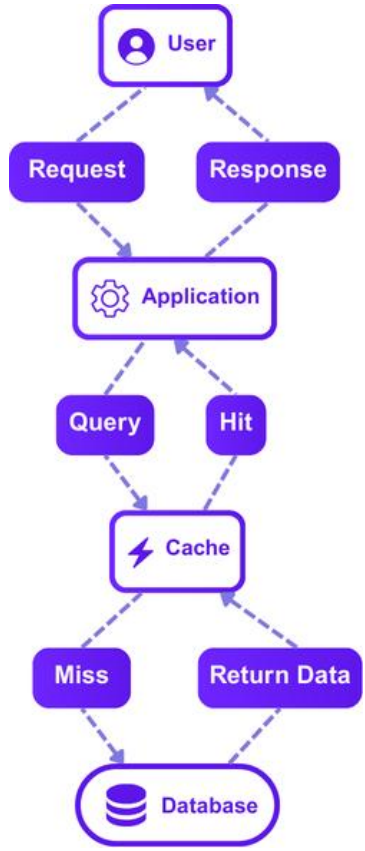
Bottlenecks:



Points in the system that slow down performance.

- **Example**: A single database server receiving too many requests at once.

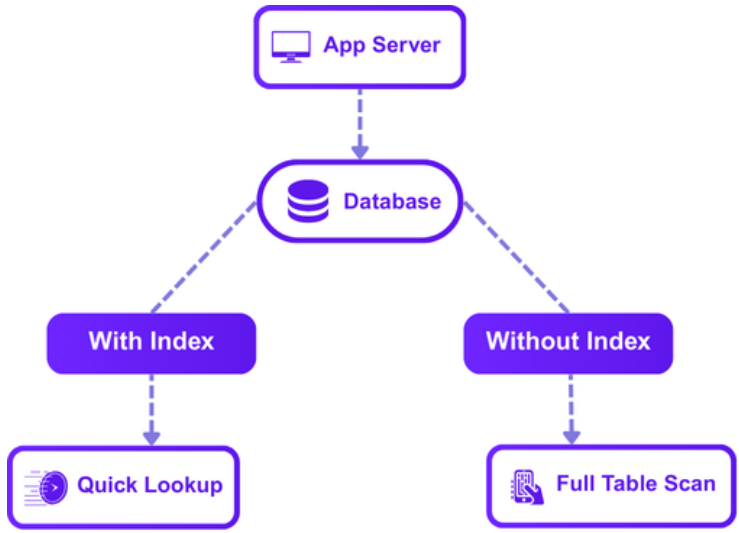
Caching:



Store frequently accessed data in memory to reduce load on databases.

- **Example:** Using Redis to cache user feeds.

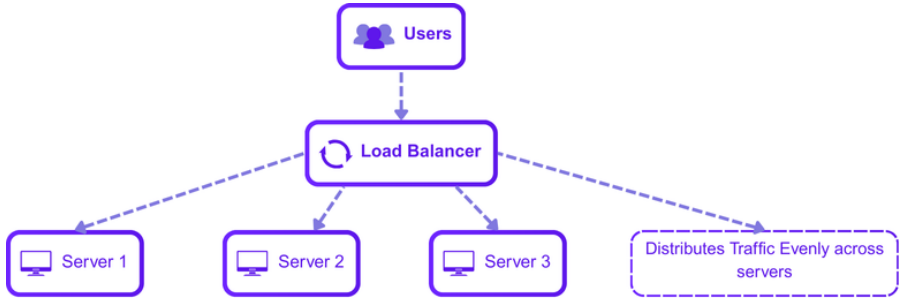
Database Indexing:



Optimize database queries by creating indexes on frequently queried fields.

- **Example**: Indexing usernames to quickly find users in a search.

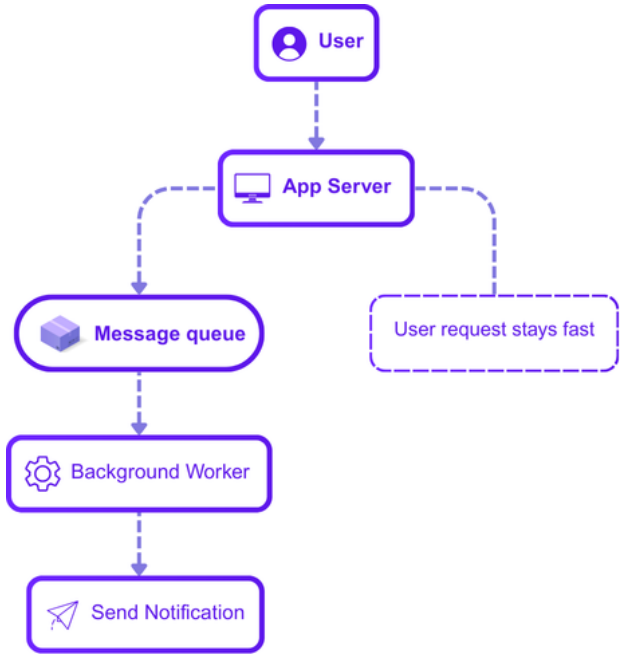
Load Balancing:



Distribute incoming requests across multiple servers to prevent any one server from becoming overwhelmed.

- **Example:** Using Nginx or AWS ELB to evenly distribute traffic.

Asynchronous Processing:



Handle long-running tasks in the background to keep the system responsive.

- **Example**: Sending email notifications via a message queue instead of during user requests.

Practical Tips:

- Identify and monitor **latency hotspots** in your system regularly.
- Use caching and indexing strategically to reduce repetitive computation and database load.
- Implement load balancing for critical services to prevent a single server from becoming a bottleneck.
- Process tasks asynchronously whenever possible to keep user-facing operations fast.
- Test system performance under realistic loads to identify potential bottlenecks before they affect users.

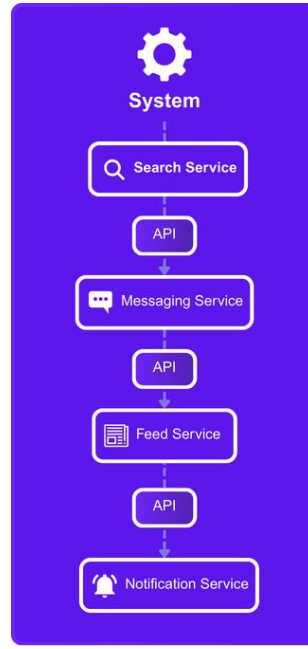
Maintainability & Extensibility

Maintainability refers to how easy it is to update, fix, or improve a system over time. Extensibility is the system's ability to add new features or expand functionality without major redesigns. Together, they ensure the system can evolve with changing requirements and continue to meet user needs.

- Reduces the cost and effort of fixing bugs or adding features.
- Makes it easier for new team members to understand the system.
- Improves system longevity and adaptability.
- Supports faster delivery of updates and new functionality.

Key Principles

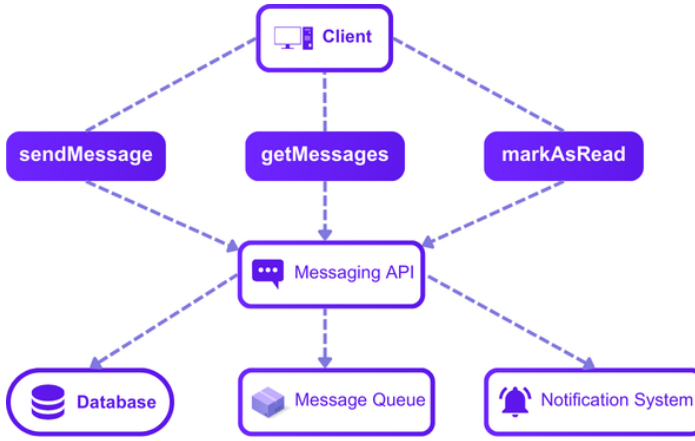
Modular Design:



Break the system into independent modules with clear responsibilities.

- **Example:** Separate services for messaging, feeds, notifications, and search.

Clear APIs:



Define consistent and well-documented interfaces between components.

- **Example:** A messaging service exposes **sendMessage()** and **getMessages()** methods with clear input/output rules.

Code Readability:

Write clean, well-structured code with descriptive names.

- **Example:** Functions and classes clearly indicate their purpose and behavior.

Documentation:

Maintain up-to-date documentation for code, APIs, and system architecture.

- **Example:** Use README files, diagrams, and API specifications for reference.