SYSTEM DESIGN

More resources :

https://github.com/InterviewReady/system-design-resources

Module 1: Basics

What is System Design?

System design is the process of defining the architecture, components, modules, interfaces, and data flow of a system to meet specific requirements. It involves creating a blueprint for building scalable, reliable, and efficient software systems. System design bridges the gap between requirements (what the system should do) and implementation (how it will be built). It encompasses both high-level design (overall architecture) and low-level design (specific components and interactions).

- **Key Aspects**:
  - Scalability: Ability to handle increased load.
  - Reliability: Ensuring the system works consistently.
  - Availability: Minimizing downtime.
  - Maintainability: Ease of updating and managing the system.
  - Performance: Optimizing response times and resource usage.
- **Why It Matters**:
  - Enables building systems that can handle millions of users (e.g., social media platforms).
  - Ensures cost-efficiency by choosing appropriate technologies.
  - Balances trade-offs like speed vs. accuracy or cost vs. performance.

Horizontal vs. Vertical Scaling

Scaling refers to increasing a system's capacity to handle more load. There are two primary approaches:

- **Horizontal Scaling**:
  - **Definition**: Adding more servers or nodes to distribute the workload.
  - **Explanation**: Instead of upgrading a single machine, you add more machines to share the load. For example, if a web server is overloaded, you add more servers behind a load balancer.
  - **Pros**:
    - Highly scalable, as you can keep adding nodes.
    - Fault-tolerant: If one server fails, others can take over.
  - **Cons**:
    - Increased complexity (e.g., managing distributed systems).
    - Requires load balancing and data synchronization.
  - **Example**: Adding more web servers to handle traffic spikes on an e-commerce site.
- **Vertical Scaling**:
  - **Definition**: Upgrading an existing server with more resources (CPU, RAM, etc.).
  - **Explanation**: You increase the power of a single machine to handle more load. For example, upgrading a server's RAM from 16GB to 64GB.
  - **Pros**:
    - Simpler to implement (no need for distributed systems).
    - Less overhead for synchronization.
  - **Cons**:
    - Limited by hardware constraints (e.g., max CPU/RAM capacity).
    - Single point of failure: If the server fails, the system goes down.
  - **Example**: Upgrading a database server's storage to handle larger datasets.

What is Capacity Estimation?

Capacity estimation is the process of calculating the resources (e.g., servers, storage, bandwidth) needed to support a system's expected workload. It ensures the system can handle current and future demands without over-provisioning (wasting resources) or under-provisioning (causing performance issues).

- **Steps in Capacity Estimation**:
  - **Identify Key Metrics**: User traffic, requests per second, data storage needs, etc.
  - **Estimate Usage Patterns**: Peak vs. average load (e.g., Black Friday traffic for e-commerce).
  - **Calculate Resource Needs**: CPU, memory, disk space, network bandwidth.
  - **Account for Growth**: Plan for future scalability (e.g., 2x or 10x user growth).
  - **Consider Redundancy**: Add buffers for failures or unexpected spikes.
- **Why It Matters**:
  - Prevents system overload during traffic spikes.
  - Optimizes costs by avoiding over-provisioning.
  - Guides infrastructure decisions (e.g., cloud vs. on-premises).

What is HTTP?

HTTP (HyperText Transfer Protocol) is the protocol used for communication on the World Wide Web. It defines how clients (e.g., browsers) request resources from servers and how servers respond. HTTP is stateless, meaning each request is independent unless explicitly managed (e.g., via cookies).

- **Key Features**:
  - **Request-Response Model**: Client sends a request (e.g., GET, POST), server responds with status codes (e.g., 200 OK, 404 Not Found).
  - **Methods**: GET (retrieve data), POST (send data), PUT (update), DELETE, etc.

    - **Statelessness**: No memory of previous requests unless using sessions/cookies.
    - **Headers**: Metadata like content type, authentication tokens.
    - **Status Codes**: Indicate success (200), errors (404, 500), redirects (301), etc.
- **Example**: When you visit a website, your browser sends a GET request to the server, which responds with HTML, CSS, and JavaScript to render the page.

What is the Internet TCP/IP Stack?

The TCP/IP stack is a set of protocols that enables communication over the internet. It organizes network communication into layers, each handling specific tasks.

- **Layers of the TCP/IP Stack**:
    - **Application Layer**: Protocols like HTTP, FTP, SMTP for user-facing services.
    - **Transport Layer**: TCP (reliable, connection-oriented) or UDP (fast, connectionless) for data transfer.
    - **Internet Layer**: IP (Internet Protocol) for addressing and routing packets (IPv4, IPv6).
    - **Link Layer**: Handles physical connections (e.g., Ethernet, Wi-Fi).
- **How It Works**:
    - Data is broken into packets, sent through layers, and reassembled at the destination.
    - Each layer adds headers to ensure reliable delivery and routing.
- **Example**: When you send an email, SMTP (application layer) formats the message, TCP (transport layer) ensures reliable delivery, IP (internet layer) routes it to the recipient's server, and Ethernet (link layer) handles physical transmission.

What Happens When You Enter Google.com?

When you type “google.com” into a browser and press Enter, several steps occur to load the webpage:

- **Steps**:
  - **DNS Resolution**: Browser queries a DNS server to convert “google.com” to an IP address (e.g., 142.250.190.78).
  - **TCP Connection**: Browser establishes a TCP connection with Google’s server (via a three-way handshake).
  - **HTTP Request**: Browser sends an HTTP GET request to the server.
  - **Server Processing**: Google’s server processes the request, fetching the webpage content.
  - **HTTP Response**: Server sends back HTML, CSS, JavaScript, and other resources.
  - **Rendering**: Browser parses the response, renders the page, and executes scripts for interactivity.
  - **Additional Requests**: Browser may fetch images, fonts, or APIs as needed.
- **Why It Matters**: Understanding this flow helps optimize latency, caching, and server performance in system design.

What are Relational Databases?

Relational databases store data in tables, where each table contains rows and columns. Data is organized using a schema, and tables are linked via keys (primary and foreign keys). They use SQL (Structured Query Language) for querying.

- **Key Features**:
  - **Structured Data**: Data follows a predefined schema (e.g., columns for user ID, name, email).
  - **ACID Transactions**: Ensures Atomicity, Consistency, Isolation, Durability.
  - **Relationships**: Tables are linked via keys (e.g., a “Users” table linked to an “Orders” table).
  - **Examples**: MySQL, PostgreSQL, Oracle Database.

- **Use Cases**:
  - Applications requiring structured data, like e-commerce or banking systems.
  - Complex queries involving joins (e.g., finding all orders for a user).

What are Database Indexes?

Database indexes are data structures that improve the speed of data retrieval in a database. They work like an index in a book, allowing the database to find rows faster without scanning the entire table.

- **How They Work**:
  - Indexes store a subset of data (e.g., a column's values) in a structure like a B-tree.
  - Queries use the index to locate data quickly, reducing disk I/O.
  - Example: Indexing the "email" column in a "Users" table speeds up searches like `SELECT * FROM Users WHERE email = 'user@example.com'`.
- **Pros**:
  - Faster query performance for reads.
  - Essential for large datasets.
- **Cons**:
  - Increased storage for index data.
  - Slower writes (inserts/updates), as indexes must be updated.

What are NoSQL Databases?

NoSQL databases are non-relational databases designed for flexibility, scalability, and handling unstructured or semi-structured data. They support various data models (e.g., key-value, document, column-family, graph).

- **Types of NoSQL Databases**:
  - **Key-Value**: Simple key-value pairs (e.g., Redis, DynamoDB).
  - **Document**: Stores JSON-like documents (e.g., MongoDB).
  - **Column-Family**: Organizes data in columns (e.g., Cassandra).

    - **Graph**: Stores relationships (e.g., Neo4j).
- **Key Features**:
    - Schema-less: Flexible data structures.
    - Horizontal Scaling: Easily distributed across multiple nodes.
    - High Performance: Optimized for specific workloads (e.g., high reads or writes).
- **Use Cases**:
    - Big data applications (e.g., real-time analytics).
    - Social media platforms with unstructured data.

What is a Cache?

A cache is a high-speed storage layer that stores frequently accessed data to reduce latency and database load. It sits between the application and the data source (e.g., database, API).

- **How It Works**:
    - Data is stored in memory (e.g., RAM) for fast access.
    - Common implementations: Redis, Memcached.
    - Cache hits: Data found in cache (fast).
    - Cache misses: Data fetched from the source and stored in cache.
- **Types**:
    - **In-Memory Cache**: Local to the application (e.g., in-process cache).
    - **Distributed Cache**: Shared across multiple servers (e.g., Redis cluster).
- **Use Cases**:
    - Caching user sessions, API responses, or database query results.
    - Reducing load on databases during traffic spikes.

What is Thrashing?

Thrashing occurs when a system's resources (e.g., CPU, memory) are overloaded, causing excessive swapping or context-switching, which degrades performance.

- **Explanation**:
  - In memory management, thrashing happens when the system swaps data between RAM and disk too frequently due to insufficient memory.
  - In multitasking systems, thrashing occurs when too many processes compete for CPU, leading to excessive context-switching.
- **Causes**:
  - Overloaded memory or CPU.
  - Poor resource allocation or scheduling.
- **Solutions**:
  - Increase memory or CPU resources.
  - Optimize application queries or processes.
  - Use caching to reduce database load.

What are Threads?

Threads are the smallest units of execution within a process. A process can have multiple threads, each executing tasks concurrently while sharing the same memory space.

- **Key Features**:
  - **Lightweight**: Threads are cheaper to create than processes.
  - **Shared Memory**: Threads within a process share memory, making communication fast.
  - **Parallelism**: Threads enable concurrent task execution (e.g., handling multiple user requests).
- **Types**:
  - **User Threads**: Managed by the application (e.g., Java threads).
  - **Kernel Threads**: Managed by the operating system.
- **Use Cases**:
  - Web servers use threads to handle multiple client requests simultaneously.
  - Multithreaded applications improve performance on multi-core CPUs.

Module 2: Load Balancing

What is Load Balancing?

Load balancing distributes incoming network traffic across multiple servers to ensure no single server is overwhelmed. It improves scalability, reliability, and performance.

- **How It Works**:
  - A load balancer (hardware or software) acts as a traffic cop, routing requests to available servers.
  - Common algorithms: Round-robin, least connections, IP hash.
- **Types**:
  - **Layer 4**: Balances based on network info (e.g., IP, port).
  - **Layer 7**: Balances based on application data (e.g., HTTP headers, URLs).
- **Benefits**:
  - Prevents server overload.
  - Increases fault tolerance (if one server fails, others take over).
  - Improves response times.
- **Examples**: Nginx, HAProxy, AWS Elastic Load Balancer.

What is Consistent Hashing?

Consistent hashing is a technique used to distribute data across a cluster of nodes in a way that minimizes data movement when nodes are added or removed.

- **Explanation**:
  - Nodes and data are mapped to a hash ring using a hash function.
  - Each data item is assigned to the nearest node on the ring (clockwise).
  - When a node is added or removed, only a small subset of data is reassigned, unlike traditional hashing.
- **Key Benefits**:

    - Scalability: Adding/removing nodes causes minimal disruption.
    - Load Distribution: Evenly spreads data across nodes.
    - Fault Tolerance: Handles node failures gracefully.
- **Real-World Scenario**:
    - **Use Case**: Distributed caching in a system like Redis.
    - **Scenario**: Imagine an e-commerce platform with a Redis cluster caching product details. Consistent hashing ensures that product data is evenly distributed across cache nodes. If a new node is added to handle increased traffic during a sale, only a small portion of the cache keys is remapped, avoiding a full cache rebuild and minimizing downtime.

What is Sharding?

Sharding is a database partitioning technique that splits a large dataset into smaller, manageable pieces (shards) stored across multiple servers.

- **Explanation**:
    - Each shard contains a subset of the data (e.g., users with IDs 1-1000 on Shard 1, 1001-2000 on Shard 2).
    - Sharding improves performance by distributing read/write operations across servers.
- **Types**:
    - **Range-Based Sharding**: Divides data by ranges (e.g., user IDs).
    - **Hash-Based Sharding**: Uses a hash function to assign data to shards.
    - **Geographical Sharding**: Splits data by location.
- **Pros**:
    - Scales horizontally to handle large datasets.
    - Reduces latency by isolating queries to specific shards.
- **Cons**:
    - Complex to implement (e.g., rebalancing shards).
    - Cross-shard queries can be slow.

- **Real-World Scenario**:
  - **Use Case**: Sharding a social media platform's user database.
  - **Scenario**: A platform like Twitter shards its user database by user ID ranges. For example, users with IDs 1-1M are stored in Shard 1, 1M-2M in Shard 2. When a user logs in, the system queries only the relevant shard, reducing latency. If a new server is added, consistent hashing can help rebalance shards with minimal data movement.

Module 3: DataStores

What are Bloom Filters?

A Bloom filter is a probabilistic data structure used to test whether an element is likely in a set. It's space-efficient but allows false positives (may say an element is present when it's not) with no false negatives (if it says an element is absent, it's definitely absent).

- **How It Works**:
  - Uses a bit array and multiple hash functions.
  - To add an element, hash it with k hash functions and set the corresponding bits to 1.
  - To check if an element exists, hash it and verify if all corresponding bits are 1.
  - False positives occur if unrelated elements set the same bits.
- **Pros**:
  - Extremely space-efficient.
  - Fast lookups (O(k) time, where k is the number of hash functions).
- **Cons**:
  - False positives are possible.
  - Cannot delete elements easily.
- **Real-World Scenario**:
  - **Use Case**: Checking if a URL is malicious in a web browser.

    - **Scenario**: Google Chrome uses a Bloom filter to check if a URL is in a database of known malicious sites. When you visit a URL, Chrome hashes it and checks the Bloom filter. If the filter says “not present,” the URL is safe. If it says “possibly present,” Chrome queries the full database, avoiding unnecessary queries for most safe URLs and saving bandwidth.

What is Data Replication?

Data replication is the process of storing copies of data across multiple servers or locations to improve availability, fault tolerance, and read performance.

- **Types**:
    - **Synchronous Replication**: Writes are applied to all replicas before confirming to the client (ensures consistency but slower).
    - **Asynchronous Replication**: Writes are applied to the primary first, then propagated to replicas (faster but risks data loss during failures).
    - **Master-Slave**: One master handles writes, slaves handle reads.
    - **Multi-Master**: Multiple nodes handle writes, requiring conflict resolution.
- **Pros**:
    - Improves fault tolerance: If one server fails, others serve data.
    - Enhances read performance by distributing read queries.
- **Cons**:
    - Increased complexity (e.g., managing consistency).
    - Higher storage and bandwidth costs.
- **Real-World Scenario**:
    - **Use Case**: Replicating a database for a global e-commerce platform.
    - **Scenario**: Amazon replicates its product catalog database across data centers in the US, Europe, and Asia. When a user in Europe searches for a product, the query hits a local replica, reducing latency. Asynchronous replication ensures the US master updates propagate to Europe, balancing speed and consistency.

How are NoSQL Databases Optimized?

NoSQL databases are optimized for scalability, flexibility, and performance with large-scale, unstructured, or semi-structured data.

- **Optimization Techniques**:
  - **Horizontal Scaling**: Distribute data across nodes using sharding or partitioning.
  - **In-Memory Storage**: Use memory for low-latency access (e.g., Redis).
  - **Denormalization**: Store data in a query-friendly format to avoid joins.
  - **Indexing**: Use specialized indexes (e.g., inverted indexes for search).
  - **Data Model Flexibility**: Support varied data types (e.g., JSON in MongoDB).
  - **Eventual Consistency**: Prioritize availability over immediate consistency in distributed setups.
- **Real-World Scenario**:
  - **Use Case**: Real-time analytics for a streaming platform.
  - **Scenario**: Netflix uses a NoSQL database like Cassandra to store user viewing data. Data is sharded across nodes to handle millions of concurrent users. Denormalized tables store user activity logs for fast queries, and eventual consistency ensures high availability during peak streaming hours.

What are Location-Based Databases?

Location-based databases are optimized for storing and querying geospatial data, such as coordinates (latitude, longitude) or geographical boundaries.

- **Key Features**:
  - **Geospatial Indexing**: Uses structures like R-trees or quad-trees for fast spatial queries.
  - **Query Types**: Find nearby points, calculate distances, or identify regions (e.g., “find restaurants within 5km”).
  - **Examples**: MongoDB (with GeoJSON), PostGIS, Redis (geospatial commands).

- **Use Cases**:
  - Ride-sharing apps (e.g., finding nearby drivers).
  - Mapping services (e.g., Google Maps).
  - Location-based advertising.
- **Real-World Scenario**:
  - **Use Case**: Ride-sharing driver matching.
  - **Scenario**: Uber uses a location-based database to store driver locations. When a user requests a ride, the system queries the database to find drivers within a 2km radius. A geospatial index ensures the query is fast, matching the user with the nearest driver in milliseconds.

Database Migrations

Database migrations are processes to update a database's schema or data while ensuring consistency and minimal downtime.

- **Types**:
  - **Schema Migration**: Change table structure (e.g., add a column).
  - **Data Migration**: Move or transform data (e.g., from SQL to NoSQL).
  - **Versioned Migrations**: Apply changes incrementally with version control.
- **Best Practices**:
  - **Backward Compatibility**: Ensure new schema works with old code.
  - **Zero Downtime**: Use techniques like blue-green deployments or rolling updates.
  - **Testing**: Validate migrations in a staging environment.
  - **Backup**: Always back up data before migrations.
- **Real-World Scenario**:
  - **Use Case**: Adding a new feature to an e-commerce platform.
  - **Scenario**: An online store adds a "wishlist" feature, requiring a new "Wishlist" table. The migration script creates the table and populates it with existing user data. A rolling migration ensures the database

remains available, and backward-compatible changes allow old app versions to function during the transition.

Module 4: Consistency vs. Availability

What is Data Consistency?

Data consistency ensures that all copies of data across a system reflect the same state. In distributed systems, consistency is challenging due to replication and network delays.

- **Types of Consistency**:
  - **Strong Consistency**: All nodes see the same data immediately after a write (e.g., synchronous replication).
  - **Eventual Consistency**: Nodes may have stale data temporarily but will converge to the same state (e.g., DynamoDB).
  - **Causal Consistency**: Preserves the order of causally related operations.
- **Trade-Offs**:
  - Strong consistency reduces availability (slower due to coordination).
  - Eventual consistency improves availability but risks stale reads.

Data Consistency Levels

Consistency levels define how strictly a system enforces data consistency during reads and writes.

- **Common Levels**:
  - **Strong Consistency**: Reads always return the latest write (e.g., relational databases with ACID).
  - **Eventual Consistency**: Reads may return stale data but eventually align (e.g., Cassandra).
  - **Read-Your-Writes**: Ensures a client sees their own writes immediately.
  - **Monotonic Reads**: Guarantees that once a client sees a value, it won't see older values.

- **Example**: In DynamoDB, you can choose consistency levels per query (e.g., strongly consistent reads for critical operations, eventually consistent for performance).

Transaction Isolation Levels

Transaction isolation levels define how transactions interact in a database, balancing consistency and performance.

- **Standard Levels (SQL)**:
  - **Read Uncommitted**: Allows dirty reads (sees uncommitted changes).
  - **Read Committed**: Prevents dirty reads but allows non-repeatable reads.
  - **Repeatable Read**: Ensures consistent reads within a transaction but may allow phantom reads.
  - **Serializable**: Highest isolation, prevents all anomalies but slowest.
- **Use Case**: A banking system uses Serializable isolation for transfers to prevent race conditions, while a logging system might use Read Committed for better performance.

Module 5: Message Queues

What is a Message Queue?

A message queue is a system that enables asynchronous communication between services by storing messages until purus:

- **Explanation**:
  - Producers send messages to a queue, consumers retrieve them later.
  - Decouples services, allowing independent processing.
  - Ensures reliable message delivery.
- **Benefits**:
  - Scalability: Consumers process messages at their own pace.
  - Fault Tolerance: Messages are not lost if a consumer fails.
- **Real-World Scenario**:

- **Use Case**: Order processing in e-commerce.
- **Scenario**: When a customer places an order, the order service sends a message to a queue (e.g., RabbitMQ). The inventory service consumes the message asynchronously to update stock, ensuring the order service isn't blocked.

What is the Publisher-Subscriber Model?

The publisher-subscriber (pub-sub) model is a messaging pattern where publishers send messages to a topic, and subscribers receive messages they're interested in without direct coupling.

- **How It Works**:
  - Publishers send messages to a topic (e.g., in Kafka or RabbitMQ).
  - Subscribers register for specific topics or message types.
  - The message broker routes messages to all relevant subscribers.
- **Benefits**:
  - Loose coupling between services.
  - Scalable: Multiple subscribers can process the same message.
- **Real-World Scenario**:
  - **Use Case**: Real-time notifications in a chat app.
  - **Scenario**: In a chat app like Slack, a user sends a message to a channel (publishes to a topic). The message broker sends the message to all channel members (subscribers), ensuring real-time delivery.

What are Event-Driven Systems?

Event-driven systems are architectures where components react to events (e.g., user actions, system changes) rather than following a sequential flow.

- **Key Components**:
  - **Events**: Discrete occurrences (e.g., a user clicking a button).
  - **Event Producers**: Generate events (e.g., a web server).
  - **Event Consumers**: Process events (e.g., a notification service).
  - **Event Bus**: Routes events (e.g., Kafka, AWS SNS).

- **Benefits**:
  - Scalability: Components operate independently.
  - Flexibility: Easy to add new event consumers.
- **Real-World Scenario**:
  - **Use Case**: IoT device monitoring.
  - **Scenario**: A smart thermostat sends temperature change events to an event bus. A cloud service consumes these events to adjust heating settings, and a mobile app consumes them to notify the user, all decoupled via the event bus.

Database as a Message Queue

Using a database as a message queue involves storing messages in a database table, which consumers poll and process.

- **How It Works**:
  - Messages are stored as rows with status (e.g., “pending,” “processed”).
  - Consumers query the table for pending messages, process them, and update the status.
  - Example: A job queue table in PostgreSQL.
- **Pros**:
  - Simple to implement with existing databases.
  - Persistent storage ensures no message loss.
- **Cons**:
  - Polling can be inefficient compared to dedicated queues.
  - Limited scalability for high-throughput systems.
- **Real-World Scenario**:
  - **Use Case**: Background job processing.
  - **Scenario**: A photo-sharing app stores image processing tasks in a database table. A worker process polls the table for new tasks, processes images (e.g., resizing), and marks them as done, leveraging the database’s reliability.

Module 6: DevOps Concepts

What is a Single Point of Failure?

A single point of failure (SPOF) is a component that, if it fails, causes the entire system to fail.

- **Explanation**:
  - SPOFs are critical components without redundancy (e.g., a single database server).
  - Eliminating SPOFs involves redundancy, replication, or failover mechanisms.
- **Examples**:
  - A single load balancer without a backup.
  - A single database without replicas.
- **Solutions**:
  - Use redundant systems (e.g., multiple load balancers).
  - Implement failover (e.g., standby database replicas).

What are Containers?

Containers are lightweight, portable units that package an application and its dependencies (e.g., libraries, runtime) to run consistently across environments.

- **How They Work**:
  - Containers use OS-level virtualization (e.g., Docker, Kubernetes).
  - Share the host OS kernel, making them more efficient than VMs.
  - Include only necessary components for the app.
- **Benefits**:
  - Portability: Run anywhere (cloud, on-premises).
  - Scalability: Easily deploy multiple containers.
  - Isolation: Each container runs independently.
- **Real-World Scenario**:

- **Use Case**: Microservices deployment.
- **Scenario**: A streaming service like Netflix deploys each microservice (e.g., recommendation engine, user authentication) in separate Docker containers. Kubernetes orchestrates these containers, ensuring scalability and fault tolerance during peak usage.

What is Service Discovery and Heartbeats?

Service discovery is the process of automatically detecting services in a distributed system. Heartbeats are periodic signals sent by services to indicate they're operational.

- **Service Discovery**:
  - Services register with a discovery system (e.g., Consul, ZooKeeper).
  - Clients query the system to find service locations (e.g., IP addresses).
- **Heartbeats**:
  - Services send regular "I'm alive" signals to the discovery system.
  - If heartbeats stop, the service is marked as unavailable.
- **Benefits**:
  - Dynamic scaling: New services are discovered automatically.
  - Fault tolerance: Failed services are removed from the registry.
- **Real-World Scenario**:
  - **Use Case**: Microservices communication.
  - **Scenario**: In a ride-sharing app, driver location services register with a Consul service discovery system. Each service sends heartbeats every 5 seconds. If a service fails, Consul removes it, ensuring the app routes requests to healthy services only.

How to Avoid Cascading Failures?

Cascading failures occur when the failure of one component triggers failures in others, leading to system-wide outages.

- **Prevention Techniques**:
  - **Circuit Breakers**: Stop sending requests to failing services.

    - **Timeouts**: Limit how long a request waits.
    - **Retries with Backoff**: Retry failed requests with increasing delays.
    - **Redundancy**: Use replicas or fallback mechanisms.
- **Real-World Scenario**:
    - **Use Case**: API failure handling.
    - **Scenario**: A payment gateway API fails, causing an e-commerce checkout service to hang. A circuit breaker detects the failure and routes requests to a backup payment provider, preventing the entire checkout system from crashing.

Anomaly Detection in Distributed Systems

Anomaly detection identifies unusual behavior in a system (e.g., traffic spikes, errors) to prevent failures or security issues.

- **Techniques**:
    - **Threshold-Based**: Flag metrics exceeding thresholds (e.g., CPU > 90%).
    - **Machine Learning**: Detect patterns deviating from normal behavior.
    - **Log Analysis**: Monitor logs for errors or unusual events.
- **Real-World Scenario**:
    - **Use Case**: Monitoring a cloud service.
    - **Scenario**: AWS CloudWatch monitors a web application's error rates. A machine learning model detects a sudden spike in 500 errors, triggering an alert to the DevOps team to investigate a faulty deployment.

Distributed Rate Limiting

Distributed rate limiting restricts the number of requests a client can make across a distributed system to prevent abuse or overload.

- **How It Works**:
    - Use a centralized or distributed counter (e.g., Redis) to track requests.
    - Enforce limits (e.g., 100 requests per minute per user).
    - Algorithms: Token bucket, leaky bucket.

- **Benefits**:
  - Prevents denial-of-service attacks.
  - Protects system resources.
- **Real-World Scenario**:
  - **Use Case**: API rate limiting.
  - **Scenario**: A weather API uses Redis to track user requests across multiple servers. If a user exceeds 1000 requests per hour, the system returns a 429 (Too Many Requests) response, ensuring fair resource allocation.

Module 7: Caching

What is Distributed Caching?

Distributed caching stores frequently accessed data across multiple nodes to improve performance and scalability.

- **How It Works**:
  - Data is stored in a distributed cache (e.g., Redis, Memcached).
  - Cache nodes are spread across servers, accessed via a client library.
  - Consistent hashing is often used to distribute data.
- **Benefits**:
  - Reduces database load.
  - Low-latency access to data.
- **Real-World Scenario**:
  - **Use Case**: Caching user profiles.
  - **Scenario**: A social media platform caches user profiles in a Redis cluster. When a user's profile is viewed, the system checks the cache first, reducing database queries and speeding up response times.

What are Content Delivery Networks?

Content Delivery Networks (CDNs) are distributed networks of servers that cache content closer to users to reduce latency.

- **How They Work**:
  - CDNs cache static content (e.g., images, videos) on edge servers worldwide.
  - Users are routed to the nearest edge server for faster delivery.
  - Examples: Cloudflare, Akamai, AWS CloudFront.
- **Benefits**:
  - Lowers latency for global users.
  - Reduces server load.
- **Real-World Scenario**:
  - **Use Case**: Video streaming.
  - **Scenario**: Netflix uses a CDN to cache popular movies on edge servers in various countries. When a user in Japan streams a movie, the CDN delivers it from a nearby server, reducing buffering time.

Write Policies

Write policies define how data is written to a cache and the underlying data store.

- **Types**:
  - **Write-Through**: Write to cache and data store simultaneously (ensures consistency, slower).
  - **Write-Back**: Write to cache first, sync to data store later (faster, risks data loss).
  - **Write-Around**: Write directly to data store, bypassing cache (avoids cache pollution).
- **Real-World Scenario**:
  - **Use Case**: Caching database updates.
  - **Scenario**: A banking app uses write-through caching for transaction data. When a user transfers money, the transaction is written to both

the cache and the database, ensuring consistency even if the cache fails.

Replacement Policies

Replacement policies determine which cache entries to remove when the cache is full.

- **Common Policies**:
  - **LRU (Least Recently Used)**: Evict the least recently accessed item.
  - **LFU (Least Frequently Used)**: Evict the least frequently accessed item.
  - **FIFO (First In, First Out)**: Evict the oldest item.
  - **Random Replacement**: Evict a random item.
- **Real-World Scenario**:
  - **Use Case**: Web page caching.
  - **Scenario**: A news website uses an LRU cache to store recently viewed articles. When the cache is full, the least recently viewed article is evicted, ensuring popular articles remain cached.

Module 8: Microservices

Microservices vs. Monoliths

Microservices and monoliths are two architectural approaches to building software.

- **Monoliths**:
  - A single, unified application containing all functionality.
  - **Pros**: Simpler development, easier debugging.
  - **Cons**: Hard to scale, deploy, or update large monoliths.
- **Microservices**:
  - Small, independent services that communicate via APIs.
  - **Pros**: Scalable, independently deployable, fault-isolated.
  - **Cons**: Complex to manage, requires robust networking.

- **Example**: A monolithic e-commerce app has one codebase for UI, backend, and database logic. A microservices-based app splits these into separate services (e.g., UI service, payment service).

How Monoliths are Migrated

Migrating a monolith to microservices involves breaking it into smaller, independent services.

- **Steps**:
  - **Identify Boundaries**: Group related functionality (e.g., user management, payments).
  - **Extract Services**: Refactor code into independent services.
  - **Define APIs**: Create clear interfaces (e.g., REST, gRPC).
  - **Use Service Discovery**: Implement discovery mechanisms (e.g., Consul).
  - **Test Incrementally**: Migrate one service at a time to minimize risk.
- **Real-World Scenario**:
  - **Use Case**: E-commerce platform migration.
  - **Scenario**: An online store migrates its monolithic app to microservices by extracting the inventory service first. It uses REST APIs to communicate with the monolith, gradually moving other components while ensuring uninterrupted service.

Module 9: API Gateways

How are APIs Designed?

API design involves creating interfaces for services to communicate effectively.

- **Best Practices**:
  - **RESTful Principles**: Use HTTP methods (GET, POST), clear endpoints (e.g., `/users/{id}`).
  - **Versioning**: Include versions (e.g., `/v1/users`) for backward compatibility.
  - **Error Handling**: Return clear status codes and messages.

    - **Authentication**: Use tokens (e.g., OAuth, JWT).
    - **Documentation**: Provide clear API docs (e.g., OpenAPI/Swagger).
- **Example**: A payment API might have endpoints like `POST /v1/payments` for creating payments and `GET /v1/payments/{id}` for retrieving payment status.

What are Asynchronous APIs?

Asynchronous APIs allow clients to send requests without waiting for an immediate response, improving scalability.

- **How They Work**:
    - Clients send a request and receive an acknowledgment (e.g., 202 Accepted).
    - The server processes the request later and notifies the client (e.g., via callbacks, webhooks, or polling).
    - Common in message queues or event-driven systems.
- **Benefits**:
    - Handles long-running tasks efficiently.
    - Improves client responsiveness.
- **Real-World Scenario**:
    - **Use Case**: Video processing API.
    - **Scenario**: A video-sharing app's API accepts video uploads asynchronously. The client receives a job ID, and the server processes the video (e.g., transcoding) in the background, notifying the client via a webhook when done.

Module 10: Authentication Mechanisms

OAuth

OAuth is an authorization framework that allows third-party apps to access user resources without sharing credentials.

- **How It Works**:

    - User authenticates with an authorization server (e.g., Google).
    - The server issues an access token to the third-party app.
    - The app uses the token to access resources (e.g., Google Drive).
- **Benefits**:
    - Secure: Tokens have limited scope and expiry.
    - User-friendly: No password sharing required.
- **Real-World Scenario**:
    - **Use Case**: Social media login.
    - **Scenario**: A user logs into a website using their Google account. OAuth grants the website an access token to retrieve the user's profile data without exposing their Google password.

Token-Based Authentication

Token-based authentication uses tokens (e.g., JWT) to verify user identity.

- **How It Works**:
    - User logs in, receives a token (signed data containing user info).
    - Client includes the token in API requests (e.g., in HTTP headers).
    - Server verifies the token's validity.
- **Benefits**:
    - Stateless: No server-side session storage needed.
    - Scalable: Works well in distributed systems.
- **Real-World Scenario**:
    - **Use Case**: Mobile app authentication.
    - **Scenario**: A mobile banking app issues a JWT upon login. The token is sent with each API request to authenticate the user, eliminating the need for server-side session tracking.

Access Control Lists and Rule Engines

Access Control Lists (ACLs) and rule engines define permissions for users or roles.

- **ACLs**:
  - List of permissions (e.g., read, write) for each user/role.
  - Example: User A can read File X, User B can write File Y.
- **Rule Engines**:
  - Define complex permission logic (e.g., “Allow access if user is in group X and location is Y”).
  - Example: AWS IAM policies.
- **Real-World Scenario**:
  - **Use Case**: Cloud storage permissions.
  - **Scenario**: A cloud storage service uses ACLs to allow specific users to access certain folders. A rule engine ensures only employees in the “Finance” group can access financial documents during business hours.

Module 11: System Design Tradeoffs

Pull vs. Push

Pull and push are two approaches for data transfer between systems.

- **Pull**:
  - Client requests data periodically (e.g., polling an API).
  - **Pros**: Simple, client-controlled.
  - **Cons**: Can waste resources if polling frequently.
- **Push**:
  - Server sends data to clients when available (e.g., webhooks, WebSockets).
  - **Pros**: Real-time, efficient.
  - **Cons**: Complex to implement, requires persistent connections.
- **Real-World Scenario**:
  - **Use Case**: Real-time updates in a stock trading app.

    - **Scenario**: A stock app uses WebSockets (push) to send price updates instantly, rather than the client polling the server every second, reducing latency and server load.

Memory vs. Latency

This trade-off balances memory usage with response time.

- **Memory**:
    - Using more memory (e.g., caching) reduces latency but increases costs.
    - Example: In-memory databases like Redis are fast but memory-intensive.
- **Latency**:
    - Optimizing for low memory may increase latency (e.g., disk-based storage).
    - Example: Fetching data from a hard drive is slower than RAM.
- **Real-World Scenario**:
    - **Use Case**: Search engine optimization.
    - **Scenario**: A search engine caches popular queries in memory to reduce latency. During peak traffic, it may reduce cache size to save memory, increasing latency for rare queries.

Throughput vs. Latency

Throughput is the rate of processing requests, while latency is the time per request.

- **Throughput**:
    - High throughput means handling many requests simultaneously.
    - Example: Batch processing increases throughput but may increase latency.
- **Latency**:
    - Low latency means faster individual responses.
    - Example: Real-time systems prioritize low latency over high throughput.

- **Real-World Scenario**:
  - **Use Case**: Video streaming.
  - **Scenario**: A streaming service prioritizes low latency for live sports (fast delivery of each frame) but may sacrifice throughput (fewer concurrent streams) to ensure smooth playback.

Consistency vs. Availability

The CAP theorem states that a distributed system cannot guarantee all three of Consistency, Availability, and Partition Tolerance.

- **Consistency**:
  - All nodes show the same data (e.g., strong consistency).
  - May reduce availability during network partitions.
- **Availability**:
  - System remains operational even during failures.
  - May lead to stale data (eventual consistency).
- **Real-World Scenario**:
  - **Use Case**: Social media feed.
  - **Scenario**: A social media platform prioritizes availability by serving cached posts during a server outage, accepting temporary inconsistencies to keep the app accessible.

Latency vs. Accuracy

This trade-off balances response speed with result correctness.

- **Latency**:
  - Faster responses may sacrifice accuracy (e.g., approximate results).
  - Example: A search engine returning quick results with lower relevance.
- **Accuracy**:
  - Slower, thorough processing ensures precise results.
  - Example: A recommendation engine taking longer to compute personalized results.

- **Real-World Scenario**:
  - **Use Case**: Real-time search suggestions.
  - **Scenario**: A search engine provides instant suggestions using a simple algorithm to reduce latency, even if some suggestions are less accurate, improving user experience.

SQL vs. NoSQL Databases

SQL and NoSQL databases serve different needs based on data structure and scalability.

- **SQL**:
  - Structured, schema-based, ACID-compliant.
  - Best for complex queries, structured data (e.g., financial systems).
  - Example: MySQL, PostgreSQL.
- **NoSQL**:
  - Flexible, schema-less, horizontally scalable.
  - Best for unstructured data, high throughput (e.g., social media).
  - Example: MongoDB, Cassandra.
- **Real-World Scenario**:
  - **Use Case**: E-commerce vs. social media.
  - **Scenario**: An e-commerce platform uses PostgreSQL for structured order data and complex joins, while a social media app uses MongoDB to store flexible user profiles and scale horizontally.

SYSTEM DESIGN IMPLEMENTATIONS :

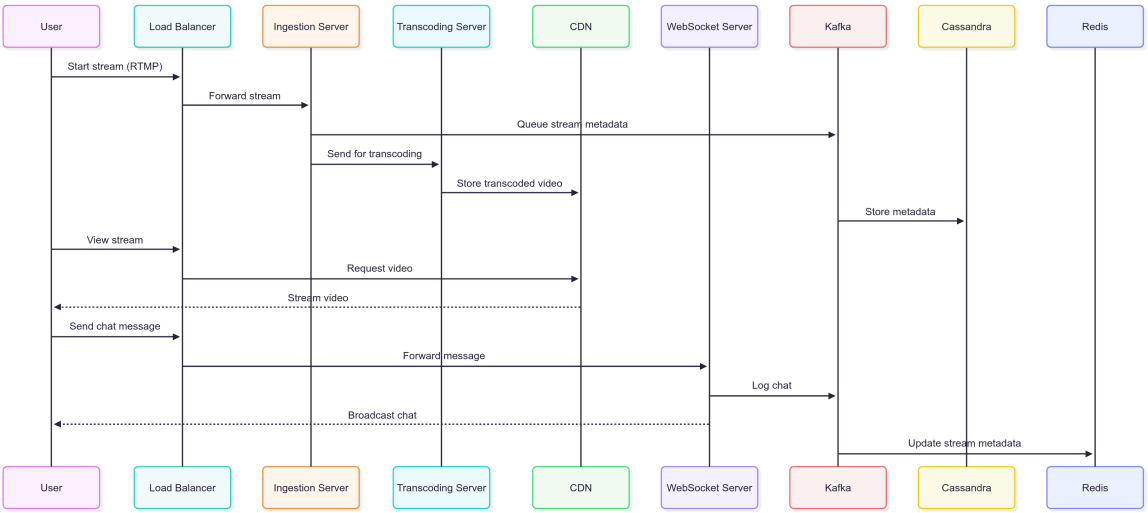
- LIVE STREAMING APP :

A live-streaming app (e.g., Twitch) enables users to broadcast and watch real-time video, supporting features like chat, subscriptions, and low-latency streaming.

- **Functional Requirements**:
  - Users can start/stop live streams.
  - Viewers can watch streams, comment, and like.
  - Support for subscriptions and notifications.
- **Non-Functional Requirements**:
  - Low latency for real-time streaming (<1s).
  - Scalability for millions of concurrent viewers.
  - High availability and fault tolerance.
- **Key Components**:
  - **Video Ingestion**: Streamers upload video via RTMP (Real-Time Messaging Protocol) to ingestion servers.
  - **Transcoding**: Convert videos to multiple resolutions (e.g., 1080p, 720p) using FFmpeg on dedicated servers.
  - **Content Delivery Network (CDN)**: Cache and deliver streams globally (e.g., AWS CloudFront).

  - **WebSockets**: Enable real-time chat and notifications.
  - **Database**: Store user profiles, stream metadata (Cassandra for scalability), and chat logs.
  - **Message Queue**: Kafka for processing stream events (e.g., likes, comments).
- **Architecture**:
  - **Ingestion**: Streamers send video to ingestion servers, which validate and forward to transcoding.

    - **Transcoding**: Videos are processed asynchronously for different devices/networks.
    - **Distribution**: CDNs cache streams, reducing latency for viewers.
    - **Chat System**: WebSockets handle real-time chat, with Kafka logging interactions.
    - **Load Balancing**: Distribute traffic across ingestion and transcoding servers.
- **Scalability**:
    - Horizontal scaling for ingestion and transcoding servers.
    - Sharding database by user/stream ID.
    - Distributed caching (Redis) for stream metadata.
- **Real-World Considerations**:
    - Use adaptive bitrate streaming to handle varying network conditions.
    - Implement circuit breakers to prevent cascading failures during server overloads.



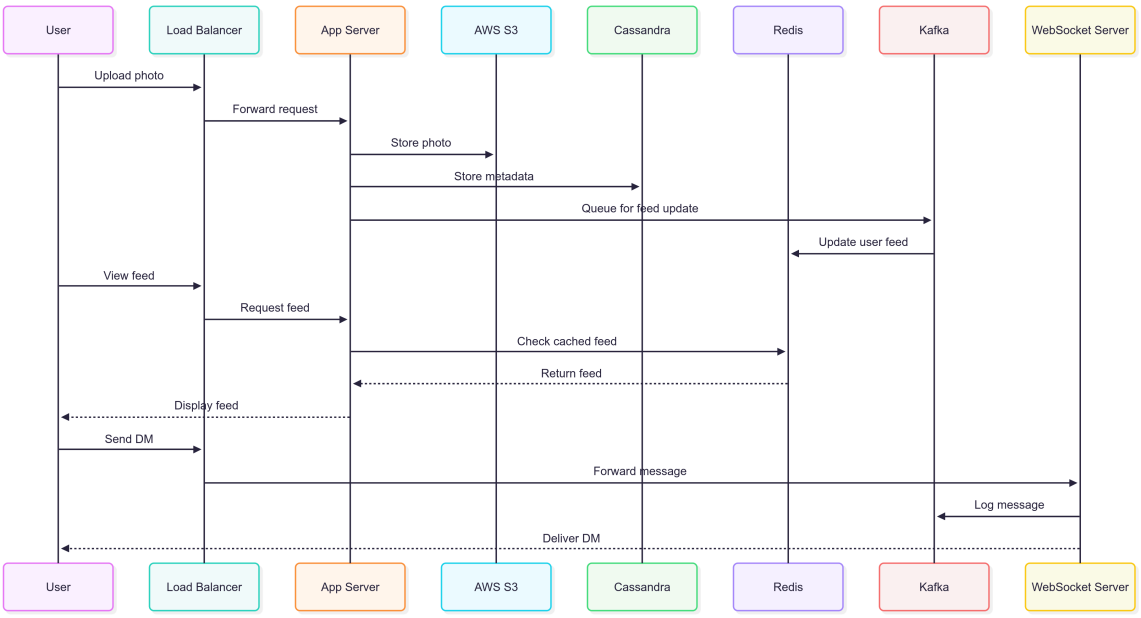
- INSTAGRAM :

Instagram is a photo-sharing platform supporting image uploads, feeds, stories, and direct messaging.

- **Functional Requirements**:
  - Upload/view photos and stories.
  - Follow users and view personalized feeds.
  - Direct messaging and likes/comments.
- **Non-Functional Requirements**:
  - High availability for global users.
  - Low latency for feed loading.
  - Scalability for millions of posts daily.
- **Key Components**:
  - **Application Servers**: Handle API requests (e.g., Django, Node.js).
  - **Database**: Cassandra for posts (NoSQL, scalable), PostgreSQL for user data (structured).
  - **Object Storage**: AWS S3 for photos/videos.

  - **CDN**: Cache images/videos for fast delivery.
  - **Cache**: Redis for feed caching and session management.
  - **Message Queue**: Kafka for processing likes, comments, and notifications.
- **Architecture**:
  - **Upload**: Users upload media to S3 via API; metadata stored in Cassandra.
  - **Feed Generation**: Precompute feeds using a fan-out-on-write approach (store user feeds in Redis).
  - **Messaging**: WebSockets for real-time DMs, Kafka for message queuing.
  - **Load Balancing**: Nginx for distributing API requests.

- **Scalability**:
  - Shard Cassandra by user/post ID.
  - Use consistent hashing for cache distribution.
  - Horizontal scaling for application servers.
- **Real-World Considerations**:
  - Optimize feed generation with eventual consistency for scalability.
  - Use Bloom filters to check if a user has already liked a post, reducing database load.

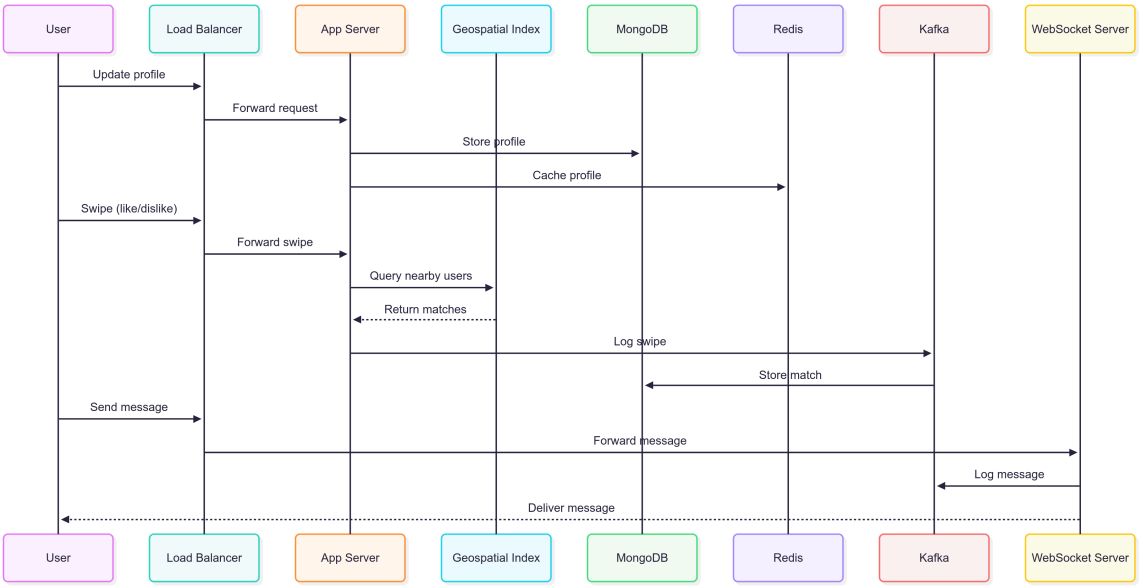


- TINDER :

Tinder is a location-based dating app for user matching, swiping, and messaging.

- **Functional Requirements**:
  - User profiles and swipe-based matching.

    - Real-time messaging for matches.
    - Location-based recommendations.
- **Non-Functional Requirements**:
    - Low latency for swipe and match operations.
    - Scalability for millions of users.
    - High availability and security.
- **Key Components**:
    - **Application Servers**: Handle swipes, matches, and profile updates.
    - **Database**: MongoDB for user profiles (flexible schema), Redis for session data.
    - **Geospatial Index**: Elasticsearch or Redis for location-based queries.
    - **Message Queue**: Kafka for match notifications and messaging.
    - **WebSockets**: Real-time chat for matched users.
- **Architecture**:
    - **Profile Management**: Users update profiles stored in MongoDB.
    - **Matching**: Geospatial index finds nearby users; swipe data processed via Kafka.
    - **Messaging**: WebSockets handle chats, with messages stored in MongoDB.
    - **Load Balancing**: Distribute traffic across servers using consistent hashing.
- **Scalability**:
    - Shard MongoDB by user ID.
    - Use distributed caching for profile data.
    - Scale matching engine horizontally.
- **Real-World Considerations**:
    - Use rate limiting to prevent swipe abuse.
    - Implement optimistic concurrency for swipe updates to avoid race conditions.

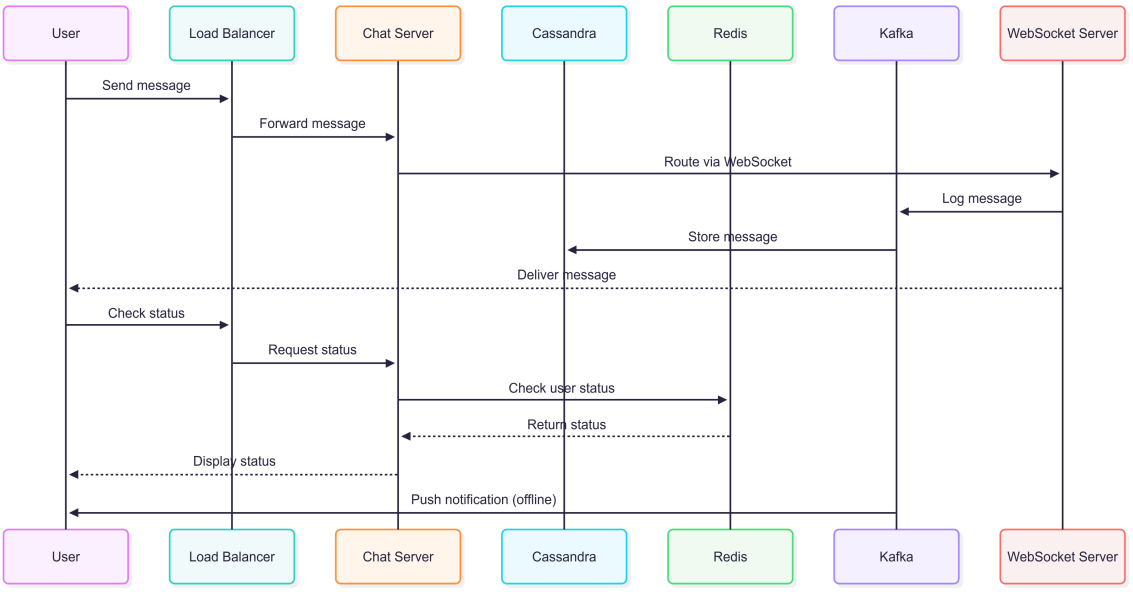


- WHATSAPP :

WhatsApp is a real-time messaging app supporting text, media, group chats, and end-to-end encryption.

- **Functional Requirements**:
  - Send/receive text, images, and videos.
  - Group chats and read receipts.
  - End-to-end encryption.
- **Non-Functional Requirements**:
  - Low latency for message delivery (<100ms).
  - High availability for billions of messages.
  - Scalability for global users.
- **Key Components**:
  - **Chat Servers**: Handle message routing (Erlang for concurrency).
  - **Database**: Cassandra for message storage, Redis for user status.
  - **Message Queue**: Kafka for asynchronous message processing.

    - **WebSockets**: Real-time message delivery and read receipts.
    - **Encryption**: End-to-end encryption using Signal Protocol.
- **Architecture**:
    - **Messaging**: WebSockets deliver messages; Kafka queues messages for offline users.
    - **Storage**: Messages stored in Cassandra with sharding by user ID.
    - **Notifications**: Push notifications via APNs/GCM for offline users.
    - **Load Balancing**: Distribute traffic across chat servers.
- **Scalability**:
    - Horizontal scaling for chat servers.
    - Shard Cassandra by user or conversation ID.
    - Use distributed caching for user status.
- **Real-World Considerations**:
    - Eventual consistency for group chats to prioritize availability.
    - Optimize storage with message compression and expiration.



- TIKTOK :

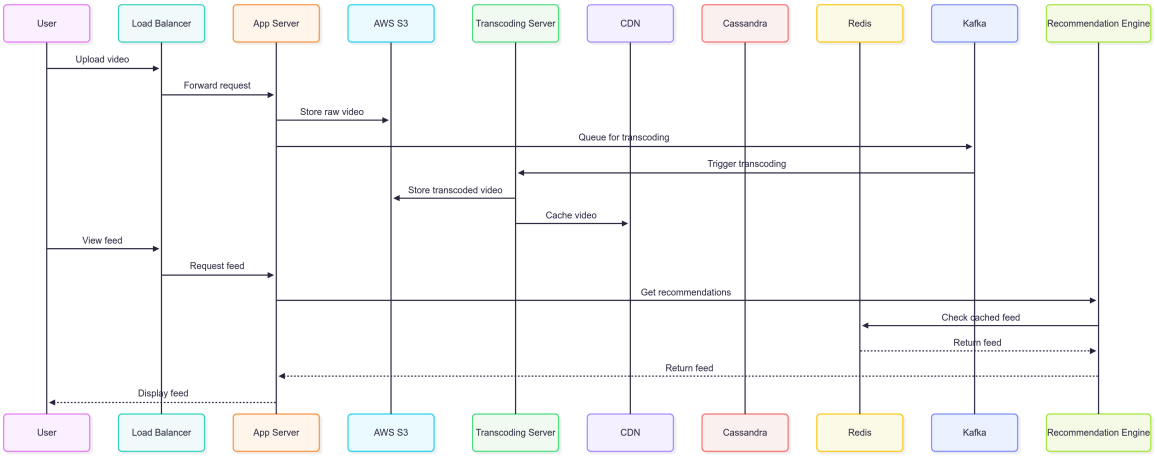
TikTok is a short-video platform for uploading, streaming, and personalized recommendations.

- **Functional Requirements**:
  - Upload and stream short videos.
  - Personalized “For You” feed.
  - Likes, comments, and follows.
- **Non-Functional Requirements**:
  - Low latency for video streaming and feed loading.
  - Scalability for billions of videos.
  - High availability and fault tolerance.
- **Key Components**:
  - **Video Storage**: AWS S3 for videos, Cassandra for metadata.

  - **CDN**: Cache videos for low-latency streaming.
  - **Recommendation Engine**: Uses Kafka and Spark for real-time processing.
  - **Database**: MongoDB for user profiles, Redis for caching.
  - **Message Queue**: Kafka for likes, comments, and feed updates.
- **Architecture**:
  - **Upload**: Videos uploaded to S3, transcoded for multiple resolutions.
  - **Feed Generation**: Machine learning models process user interactions via Kafka, store recommendations in Redis.
  - **Streaming**: CDNs deliver videos; WebSockets handle real-time interactions.
  - **Load Balancing**: Distribute traffic across upload and recommendation servers.
- **Scalability**:

- Shard Cassandra by video ID.
- Horizontal scaling for transcoding and recommendation engines.
- Use consistent hashing for cache distribution.

- **Real-World Considerations**:
  - Use adaptive bitrate streaming for varying network conditions.
  - Implement Bloom filters to deduplicate video views.

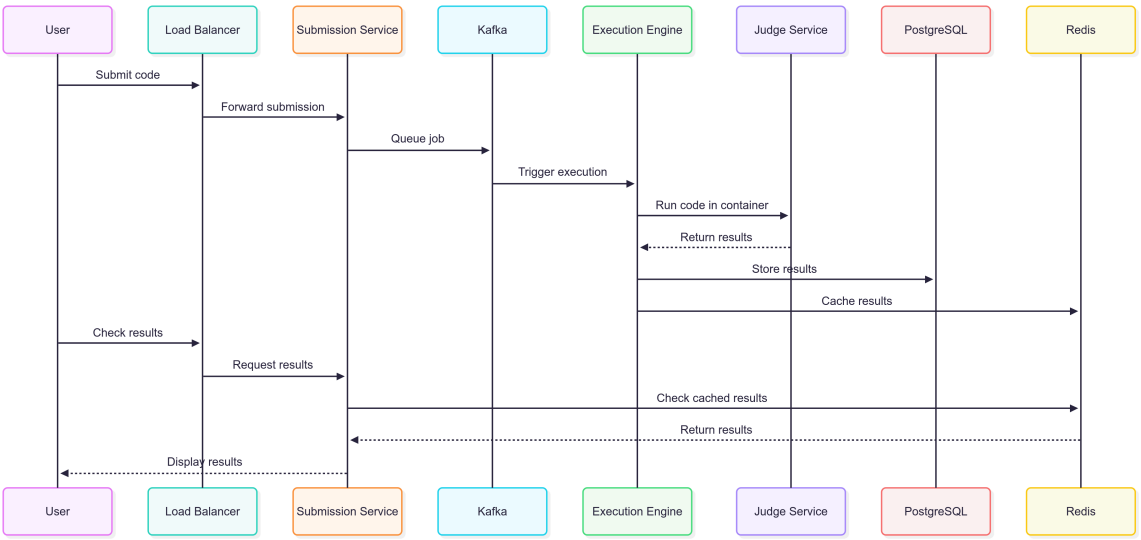


- ONLINE CODING JUDGE :

An online coding judge (e.g., LeetCode) evaluates user-submitted code in a secure, scalable environment.

- **Functional Requirements**:
  - Submit code and run against test cases.
  - Display results (pass/fail, runtime, memory).
  - Support multiple languages and problem sets.
- **Non-Functional Requirements**:
  - Low latency for code execution (<1s).
  - Scalability for thousands of submissions.

    - Security to prevent malicious code.
- **Key Components**:
    - **Submission Service**: Accepts code submissions via API.
    - **Job Queue**: Kafka queues submissions for processing.
    - **Execution Engine**: Containers (Docker) run code in isolated sandboxes.
    - **Database**: PostgreSQL for problems/users, Redis for results caching.
    - **Judge Service**: Compares output with expected results.
- **Architecture**:
    - **Submission**: Users submit code to API; Kafka queues jobs.
    - **Execution**: Containers execute code with resource limits (CPU, memory).
    - **Result Storage**: Results cached in Redis, stored in PostgreSQL.
    - **Load Balancing**: Distribute submissions across execution nodes.
- **Scalability**:
    - Horizontal scaling for execution nodes.
    - Shard PostgreSQL by problem/user ID.
    - Use distributed caching for results.
- **Real-World Considerations**:
    - Use circuit breakers to handle execution timeouts.
    - Implement rate limiting to prevent submission abuse.



- UPI PAYMENTS :

Unified Payments Interface (UPI) is a real-time payment system for instant money transfers.

- **Functional Requirements**:
  - Send/receive money using UPI IDs.
  - Transaction history and notifications.
  - Bank integration for fund transfers.
- **Non-Functional Requirements**:
  - High availability for 24/7 transactions.
  - Low latency (<100ms).
  - Strong consistency for financial transactions.
- **Key Components**:
  - **Payment Service**: Handles transaction requests (e.g., Spring Boot).
  - **Database**: PostgreSQL for transaction records (ACID-compliant).
  - **Message Queue**: Kafka for transaction logging and notifications.
    - UPI PAYMENTS :
  - **Bank Integration**: NPCI APIs for inter-bank transfers.

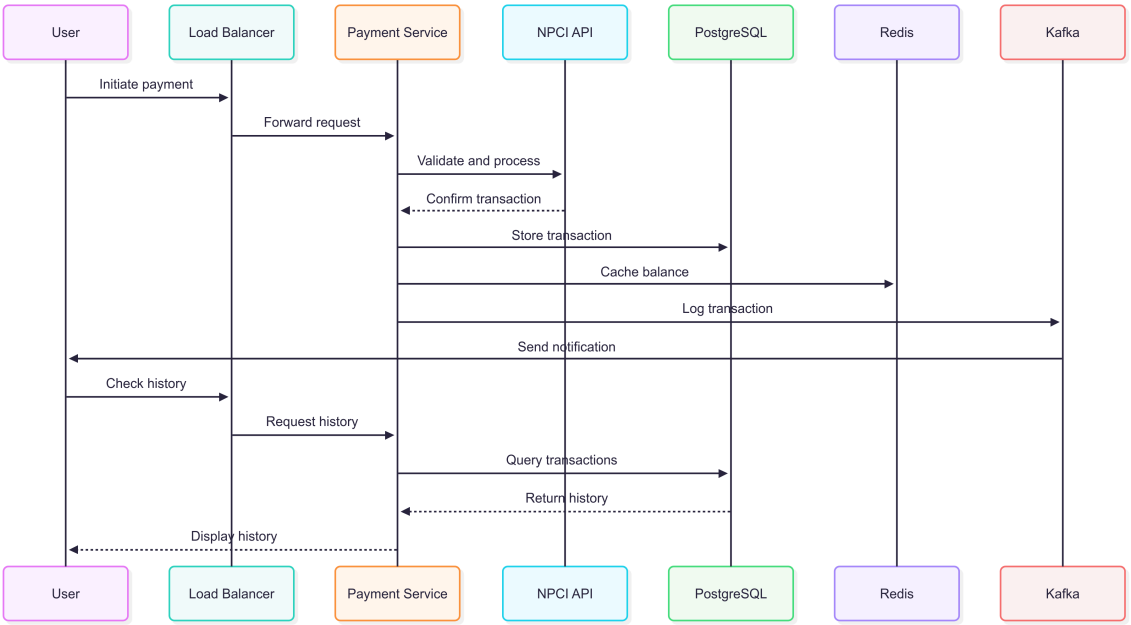
    - **Cache**: Redis for user session and balance caching.
- **Architecture**:
    - **Transaction Flow**: User initiates transfer; payment service validates via NPCI.
    - **Storage**: Transactions stored in PostgreSQL with strong consistency.
    - **Notifications**: Kafka triggers SMS/email notifications.
    - **Load Balancing**: Distribute requests across payment servers.
- **Scalability**:
    - Horizontal scaling for payment servers.
    - Shard PostgreSQL by user ID.
    - Use distributed rate limiting to prevent abuse.
- **Real-World Considerations**:
    - Use optimistic locking for transaction concurrency.
    - Implement anomaly detection for fraud prevention.

- IRCTC :

IRCTC is an online railway ticket booking system for managing reservations and availability.

- **Functional Requirements**:
    - Search trains and book tickets.
    - Check seat availability and payment processing.
    - Cancellation and refund processing.
- **Non-Functional Requirements**:
    - High availability during peak booking hours.
    - Scalability for millions of users.
    - Strong consistency for seat allocation.

- **Key Components**:
  - **Booking Service**: Handles search and booking APIs.
  - **Database**: PostgreSQL for train schedules and bookings (ACID).
  - **Cache**: Redis for caching train availability.
  - **Payment Gateway**: Integrates with banks for payments.
  - **Message Queue**: Kafka for booking confirmations and refunds.
- **Architecture**:
  - **Search**: Users query train schedules cached in Redis.
  - **Booking**: Locks seats using optimistic concurrency; stores in PostgreSQL.
  - **Payments**: Integrates with payment gateways; Kafka logs transactions.
  - **Load Balancing**: Distribute traffic across booking servers.
- **Scalability**:
  - Shard PostgreSQL by train/route ID.
  - Horizontal scaling for booking servers.
  - Use consistent hashing for cache.
- **Real-World Considerations**:
  - Use circuit breakers for payment gateway failures.
  - Implement rate limiting for booking requests during peak hours.



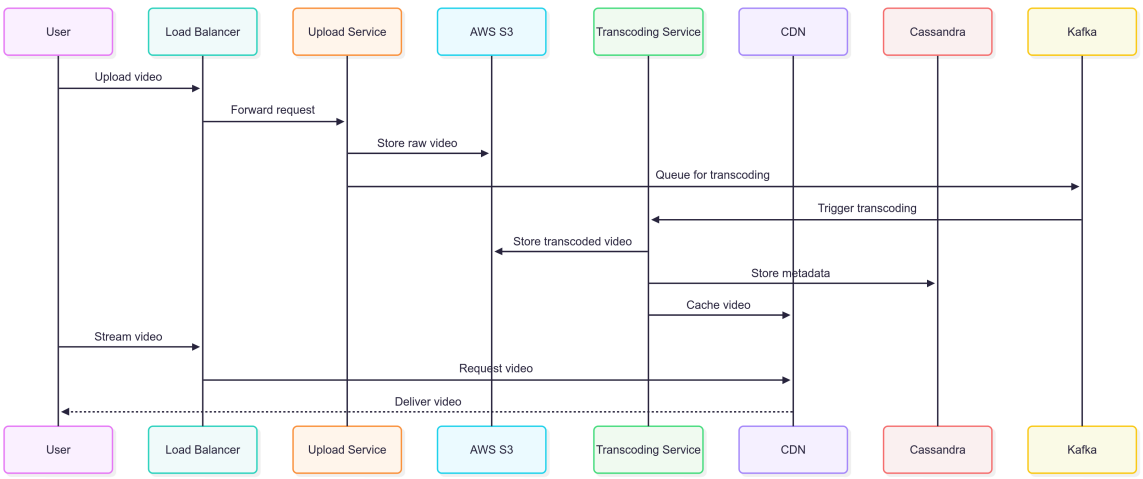
- NETFLIX VIDEO ONBORADING PIPELINE :

Netflix's video onboarding pipeline processes and stores uploaded videos for streaming.

- **Functional Requirements**:
  - Upload and transcode videos.
  - Store videos and metadata.
  - Support multiple formats for streaming.
- **Non-Functional Requirements**:
  - Scalability for millions of video uploads.
  - Low latency for transcoding and delivery.
  - High durability for video storage.
- **Key Components**:
  - **Upload Service**: Accepts video uploads via API.
  - **Object Storage**: AWS S3 for raw and transcoded videos.

- **Transcoding Service**: FFmpeg for converting videos to multiple resolutions.
- **Message Queue**: Kafka for queuing transcoding jobs.
- **CDN**: Open Connect (Netflix's CDN) for video delivery.

- **Architecture**:
  - **Upload**: Videos uploaded to S3; metadata stored in Cassandra.
  - **Transcoding**: Kafka queues jobs; containers process videos.
  - **Delivery**: CDNs cache videos for streaming.
  - **Load Balancing**: Distribute transcoding jobs across nodes.
- **Scalability**:
  - Horizontal scaling for transcoding nodes.
  - Shard Cassandra by video ID.
  - Use distributed caching for metadata.
- **Real-World Considerations**:
  - Use asynchronous processing for transcoding to handle large files.
  - Implement lifecycle policies in S3 to archive old videos.



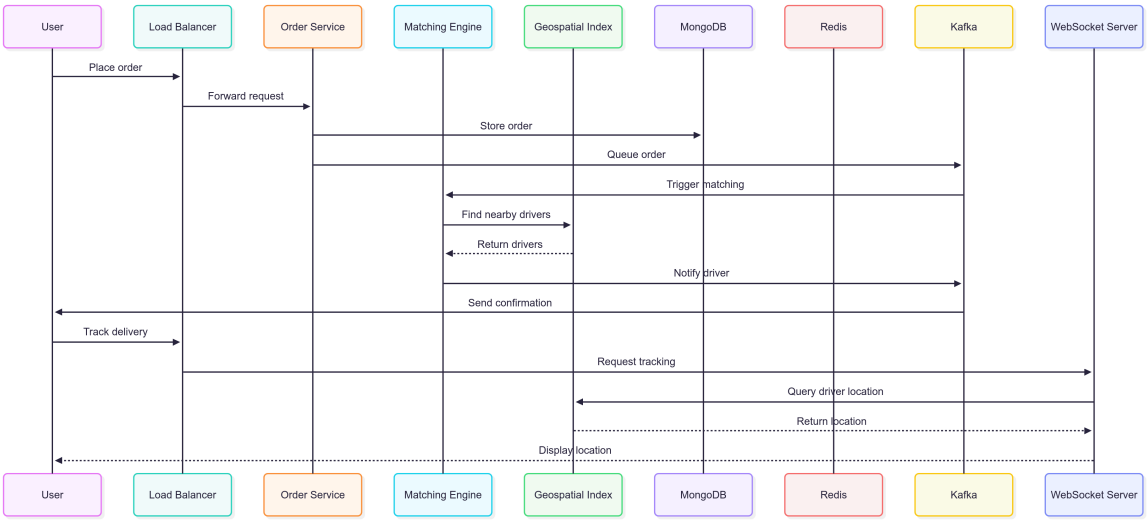
- DOORDASH :

DoorDash is a food delivery platform connecting customers, restaurants, and drivers.

- **Functional Requirements**:
  - Browse restaurants and place orders.
  - Match orders with delivery drivers.
  - Real-time tracking and payments.
- **Non-Functional Requirements**:
  - Low latency for order placement and tracking.
  - Scalability for peak demand.
  - High availability and reliability.
- **Key Components**:
  - **Order Service**: Handles order placement and restaurant menus.
  - **Matching Engine**: Assigns drivers based on proximity (geospatial index).
  - **Database**: MongoDB for orders, Redis for caching.
  - **Message Queue**: Kafka for order updates and notifications.
  - **Geospatial Index**: Elasticsearch for driver location tracking.
- **Architecture**:
  - **Order Placement**: Users browse menus cached in Redis; orders stored in MongoDB.
  - **Driver Matching**: Geospatial index matches drivers; Kafka notifies drivers.
  - **Tracking**: WebSockets provide real-time location updates.
  - **Load Balancing**: Distribute traffic across order and matching servers.
- **Scalability**:
  - Shard MongoDB by order/restaurant ID.

- Horizontal scaling for matching engines.
- Use consistent hashing for cache.

- **Real-World Considerations**:
  - Use circuit breakers for payment gateway failures.
  - Implement surge pricing for peak demand.

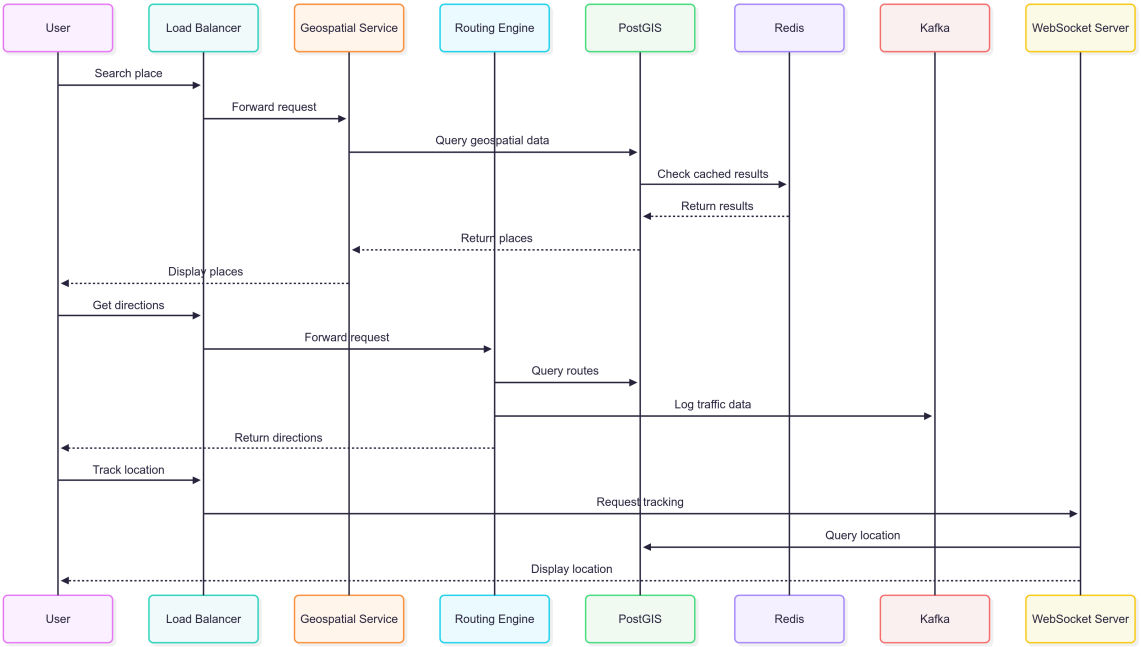


- GOOGLE MAPS :

Google Maps provides location-based services like navigation, search, and real-time traffic.

- **Functional Requirements**:
  - Search places and get directions.
  - Real-time traffic updates and ETA.
  - Location-based recommendations.
- **Non-Functional Requirements**:
  - Low latency for search and navigation (<100ms).
  - Scalability for billions of queries.

    - High availability and accuracy.
- **Key Components**:
    - **Geospatial Service**: Handles place search and routing.
    - **Database**: PostGIS for geospatial data, Redis for caching.
    - **Routing Engine**: Calculates optimal routes using graph algorithms.
    - **Message Queue**: Kafka for traffic updates.
    - **Cache**: Redis for frequent searches and routes.
- **Architecture**:
    - **Search**: Geospatial index (PostGIS) finds places; results cached in Redis.
    - **Routing**: Graph-based algorithms compute routes; Kafka processes traffic data.
    - **Tracking**: WebSockets provide real-time location updates.
    - **Load Balancing**: Distribute queries across geospatial servers.
- **Scalability**:
    - Shard PostGIS by region.
    - Horizontal scaling for routing engines.
    - Use consistent hashing for cache.
- **Real-World Considerations**:
    - Use Bloom filters to check cached routes.
    - Implement rate limiting for API queries.

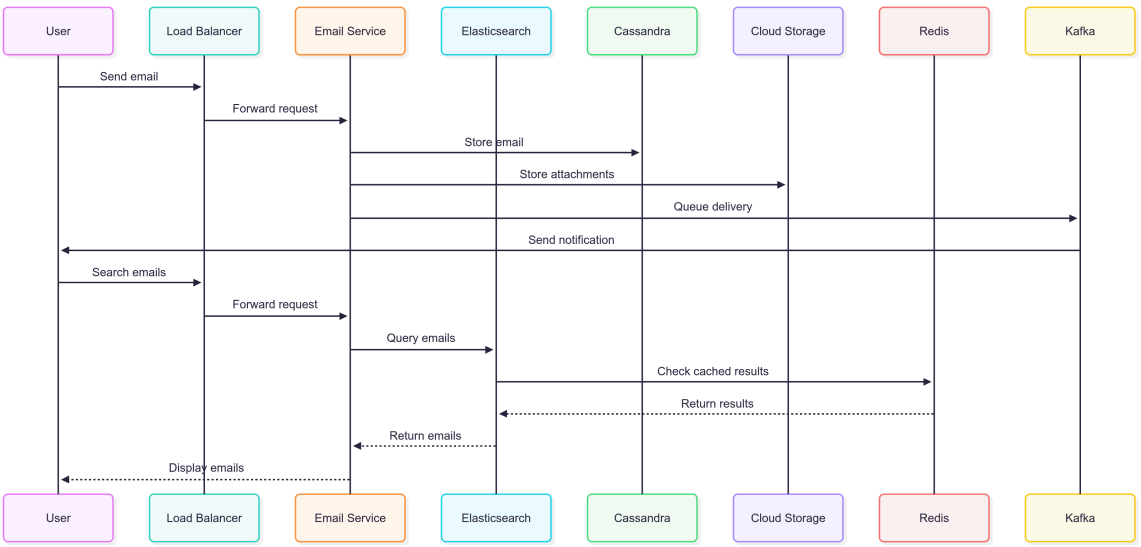


- GMAIL :

Gmail is an email service supporting sending, receiving, and searching emails.

- **Functional Requirements**:
  - Send/receive emails.
  - Search emails and manage labels.
  - Attachments and spam filtering.
- **Non-Functional Requirements**:
  - High availability for global users.
  - Scalability for billions of emails.
  - Low latency for search and delivery.
- **Key Components**:
  - **Email Service**: Handles sending/receiving emails (SMTP, IMAP).
  - **Database**: Cassandra for email storage, Elasticsearch for search.
  - **Object Storage**: Google Cloud Storage for attachments.
  - **Cache**: Redis for user sessions and frequent searches.

    - **Message Queue**: Kafka for email processing and notifications.
- **Architecture**:
    - **Sending/Receiving**: SMTP servers handle email delivery; Cassandra stores emails.
    - **Search**: Elasticsearch indexes emails for fast search.
    - **Attachments**: Stored in Cloud Storage; metadata in Cassandra.
    - **Load Balancing**: Distribute traffic across email servers.
- **Scalability**:
    - Shard Cassandra by user ID.
    - Horizontal scaling for email and search servers.
    - Use distributed caching for search results.
- **Real-World Considerations**:
    - Use eventual consistency for non-critical operations (e.g., spam filtering).
    - Implement rate limiting for email sending.

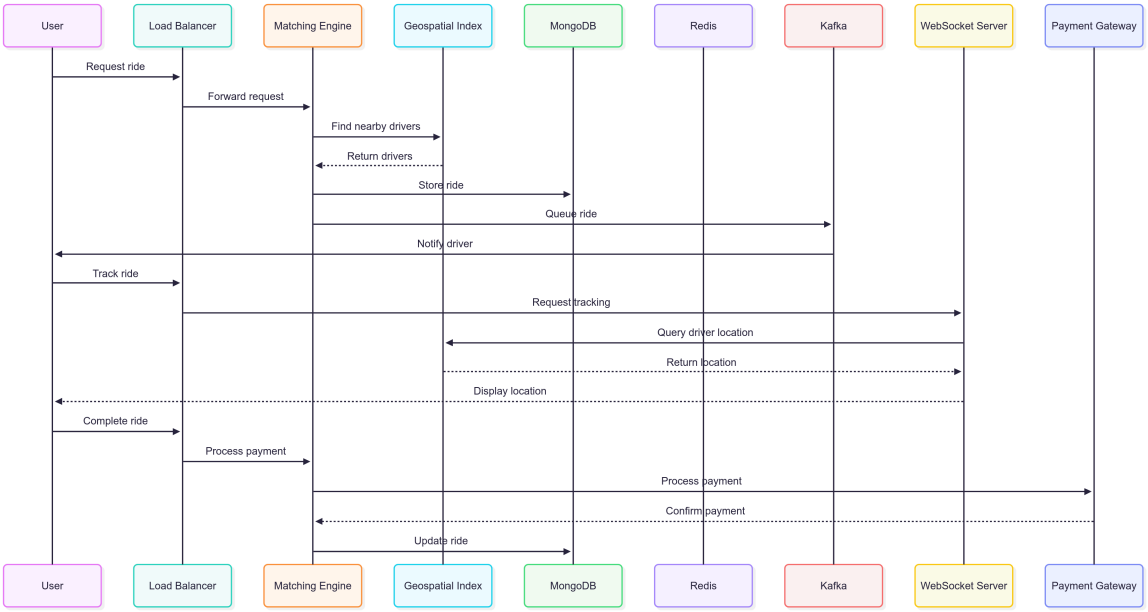


- UBER :

Uber is a ride-sharing platform connecting riders and drivers with real-time tracking.

- **Functional Requirements**:
  - Request rides and match with drivers.
  - Real-time tracking and ETA.
  - Payments and ratings.
- **Non-Functional Requirements**:
  - Low latency for matching and tracking (<100ms).
  - Scalability for millions of rides.
  - High availability and reliability.
- **Key Components**:
  - **Matching Engine**: Assigns drivers based on proximity (geospatial index).
  - **Database**: MongoDB for ride data, Redis for caching.
  - **Geospatial Index**: PostGIS or Elasticsearch for location queries.
  - **Message Queue**: Kafka for ride updates and notifications.
  - **WebSockets**: Real-time tracking and driver communication.
- **Architecture**:
  - **Ride Request**: Users request rides; matching engine finds drivers.
  - **Tracking**: WebSockets provide real-time location updates.
  - **Payments**: Integrates with payment gateways; Kafka logs transactions.
  - **Load Balancing**: Distribute traffic across matching servers.
- **Scalability**:
  - Shard MongoDB by ride ID.
  - Horizontal scaling for matching and tracking servers.
  - Use consistent hashing for cache.

- **Real-World Considerations**:
  - Use circuit breakers for payment gateway failures.
  - Implement surge pricing for peak demand.

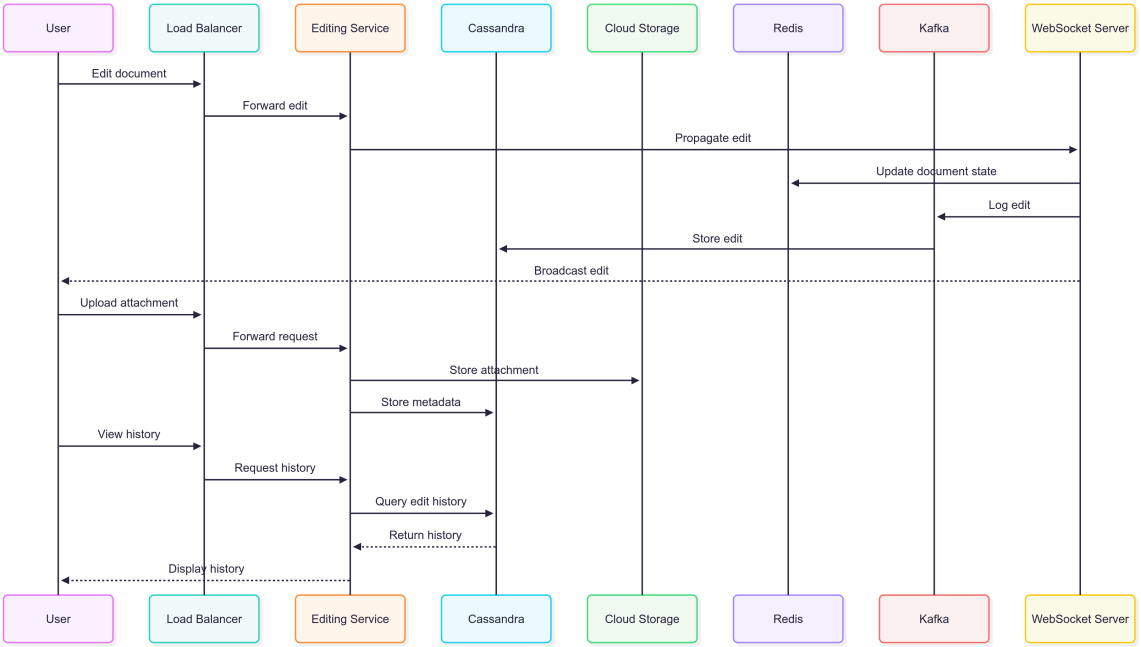


- GOOGLE DOCS :

Google Docs is a collaborative document-editing platform supporting real-time editing and sharing.

- **Functional Requirements**:
  - Real-time collaborative editing.
  - Document storage and sharing.
  - Version history and comments.
- **Non-Functional Requirements**:
  - Low latency for edits (<100ms).
  - Scalability for millions of documents.

    - Strong consistency for edits.
- **Key Components**:
    - **Editing Service**: Handles real-time edits (Operational Transformation or CRDT).
    - **Database**: Cassandra for document storage, Redis for active sessions.
    - **WebSockets**: Real-time edit propagation.
    - **Object Storage**: Google Cloud Storage for large documents.
    - **Message Queue**: Kafka for edit logs and notifications.
- **Architecture**:
    - **Editing**: WebSockets propagate edits; CRDT ensures conflict-free updates.
    - **Storage**: Documents stored in Cassandra; attachments in Cloud Storage.
    - **Versioning**: Kafka logs edit history for rollback.
    - **Load Balancing**: Distribute traffic across editing servers.
- **Scalability**:
    - Shard Cassandra by document ID.
    - Horizontal scaling for editing servers.
    - Use distributed caching for active documents.
- **Real-World Considerations**:
    - Use optimistic concurrency for edit conflicts.
    - Implement rate limiting for edit requests to prevent abuse.


User
Load Balancer
Editing Service
Cassandra
Cloud Storage
Redis
Kafka
WebSocket Server
Edit document
Forward edit
Propagate edit
Update document state
Log edit
Store edit
Broadcast edit
Upload attachment
Forward request
Store attachment
Store metadata
View history
Request history
Query edit history
Return history
Display history
User
Load Balancer
Editing Service
Cassandra
Cloud Storage
Redis
Kafka
WebSocket Server