HLD Microservices 1

Contents

[Services and APIs 2](#_Toc184990096)

[Monolith vs Microservices: 5](#_Toc184990097)

[Monolithic and Service-Oriented Architecture (SOA) vs. Microservices: 7](#_Toc184990098)

[API Gateway, Load Balancers, and Microservices 10](#_Toc184990099)

[Monoliths, Microservices, and API Gateway 13](#_Toc184990100)

[Microservices vs. Monolith Overview: 15](#_Toc184990101)

[Pros for Microservices 18](#_Toc184990102)

[Cons for Microservices 20](#_Toc184990103)

[Microservices Some Topics 25](#_Toc184990104)

[Observability 28](#_Toc184990105)

[Time Series Databases and Metrics Analysis 33](#_Toc184990106)

[User Engagement Metrics and Their Tracking 35](#_Toc184990107)

## Services and APIs

**Introduction to Services**

* A **service** is a combination of:
  + **Application Layer**: Processes logic and computations.
  + **Storage Layer**: Manages data, typically with databases and caching mechanisms.
* It exposes certain **APIs (Application Programming Interfaces)** to allow interaction for specific tasks, such as fetching data or updating it.

**Google Search Type-Ahead Case Study**

* **Type-Ahead Feature**:
  + When a user starts typing in the search bar, it suggests options (e.g., typing "Mich" may suggest "Michael Jackson" or "Michelle Obama").
  + Two key APIs involved:
    1. **Get Top Suggestions**: Given a prefix, returns suggestions.
    2. **Update Frequency**: Updates the frequency of searched terms.
* Services involved:
  + **Type-Ahead Service**: Handles the prefix matching and updating frequencies.
  + **Search Results Service**: Fetches actual search results based on the user's query.

**Understanding Services**

* **Components of a Service**:
  + Application servers, databases, caching layers, and possibly other components.
* A **service** is not a single server or API but the entire setup required to perform a specific function.
* Examples:
  + The **Auto-Complete Service** exposes APIs for fetching suggestions and updating frequencies.
  + The **Search Results Service** processes user queries to return relevant results.

**APIs within Services**

* APIs serve as the **interface** to interact with a service.
  + Example: getTopSuggestions(prefix) API might internally use tries, hashmaps, or databases but abstracts those details from the user.
* APIs act like **waiters** in a restaurant:
  + The service (restaurant) has various components (kitchen, chefs, ingredients).
  + APIs (waiters) communicate with the internal components and provide results to the user.
  + Users don't need to know the internal workings of the service, just the API functions they can call.

**Analogy for Better Understanding**

* A **restaurant** is akin to a service:
  + Components include the kitchen, chefs, ingredients, and other resources.
  + The waiter (API) takes the user’s request, interacts with the internal system, and delivers the output (e.g., a meal).

**Monolithic Architecture**

* **Definition**: A software architecture where all components (frontend, backend, business logic, etc.) are tightly coupled into a single codebase and deployed as a single unit.
* **Characteristics**:
  1. **Single Codebase**: All features and services are part of the same application.
  2. **Unified Deployment**: The application compiles into one executable or deployable file (e.g., a .jar file in Java).
  3. **Centralized Database**: Often uses a single or few centralized databases for all operations.

**Advantages:**

1. **Simplified Development**:
   * Easy to set up for small-scale projects.
   * Development and deployment processes are straightforward.
2. **Efficient Resource Use**:
   * No overhead of managing inter-service communication.
3. **Easier Debugging**:
   * Everything resides in a single environment, making tracing issues more linear.
4. **Best for Small Teams**:
   * Works well when the team and project size are small, with limited complexity.

**Challenges:**

1. **Scalability Issues**:
   * Difficult to scale specific components. For example, if the search functionality needs more resources, the entire application must scale.
2. **Tightly Coupled**:
   * Code changes in one area can inadvertently impact others.
   * Functions often have implicit dependencies.
3. **Onboarding Complexity**:
   * New engineers need to understand the entire codebase, which can be overwhelming for large projects.
4. **Slow Deployment**:
   * Even minor changes require recompiling and redeploying the entire application.
5. **Bug Propagation**:
   * A bug in one module (e.g., order processing) can affect unrelated parts (e.g., search results).
6. **Resource Contention**:
   * High-demand functions (e.g., slow order processing) can hog server resources, impacting other operations.
7. **Build Times**:
   * As the codebase grows, the time to build and compile the project increases significantly.

**Monolithic vs. Service-Oriented Architecture (SOA) or Microservices:**

1. **Monolithic**:
   * All functionalities in a single executable file.
   * Ideal for smaller projects or teams.
   * Deployment cycles are longer and involve the whole application.
2. **Service-Oriented/Microservices**:
   * Functionalities are broken into independent services (e.g., Search Service, AutoComplete Service).
   * Each service can be deployed, scaled, and maintained independently.
   * Suitable for complex, large-scale systems.

**Use Case Scenarios:**

* **Monolithic**: Best for startups, prototypes, or projects with a limited feature set and small teams.
* **Microservices**: Preferred for large-scale, complex systems where scalability, modularity, and independent deployments are critical (e.g., Google, Amazon).

## Monolith vs Microservices:

1. **Monolith:**
   * A single project file containing multiple components (e.g., view, model, controller).
   * Compiled into a single artifact (e.g., a .jar file).
   * Deployed as a single unit to one or more servers.
   * Even if it performs multiple tasks or exposes multiple functions, it is still a **monolith** as long as everything resides in a single project.
2. **Microservices:**
   * Separate projects with distinct files for each service.
   * Each project compiles into its own independent artifact (e.g., project1.jar, project2.jar).
   * These artifacts are deployed separately, possibly on different machines.
   * Each microservice operates independently; changes in one service don’t directly impact others.

**Example Breakdown**

* **Project 1:**
  + Files: view.html, model.java, controller.java.
  + Compiles into project1.jar.
  + Deployed on separate servers (app server 1, app server 2).
* **Project 2:**
  + Different files with independent functionality.
  + Compiles into project2.jar.
  + Deployed on different machines (machine 1, machine 2).

**Comparison**

| **Aspect** | **Monolith** | **Microservices** |
| --- | --- | --- |
| **Project Structure** | Single project file. | Multiple independent project files. |
| **Compilation** | Single artifact. | Each service has its own compiled artifact. |
| **Deployment** | Deployed as one unit. | Deployed on separate machines. |
| **Interdependence** | Tightly coupled; changes affect the whole. | Loosely coupled; changes affect individual services. |
| **Scalability** | Harder to scale individual components. | Scalable per service needs. |

## Monolithic and Service-Oriented Architecture (SOA) vs. Microservices:

1. **Monolithic Architecture**:
   * **Single Codebase**: All functionality resides within a single project, organized into separate files or modules but deployed as one unit.
   * **Shared Infrastructure**: Common app servers and a single database handle all features (e.g., user profiles, products, orders, payments).
   * **Challenges**: Scaling specific components independently is difficult. A small change might require redeploying the entire application.
2. **Service-Oriented Architecture (SOA)**:
   * **Multiple Services**: Applications are broken into services (e.g., user service, product service) that handle specific business logic.
   * **Database Sharing Allowed**: Services may share databases.
   * **Synchronous Communication**: Services often rely on synchronous requests/responses.
   * **Limitations**: While modular, tight coupling between services can lead to interdependencies and bottlenecks.
3. **Microservices Architecture**:
   * **Independent, Functionality-Driven Services**: Each microservice is responsible for a specific feature or functionality, such as user management, product catalogue, or payment processing.
   * **Dedicated Databases**: Each microservice has its own database, ensuring clear boundaries and reducing coupling.
   * **Flexible Communication**:
     + **Synchronous**: Direct calls for immediate responses.
     + **Asynchronous/Event-Driven**: Notifications via events or message queues (e.g., Kafka, RabbitMQ) for scalability and responsiveness.
   * **Trade-offs**:
     + **Data Duplication**: Allows microservices to maintain local copies of relevant data, leading to potential data inconsistency issues.
     + **Increased Complexity**: Requires mechanisms for inter-service communication, error handling, monitoring, and deployment management.

**Example Transition to Microservices: Flipkart Use Case**

**Monolith:**

* All logic (e.g., user profile, product details, cart, payment, order tracking) is in a single codebase.
* **Shared Resources**:
  + One database (e.g., MongoDB for product details).
  + Shared image storage (e.g., S3 for product images).

**Breaking into Microservices:**

1. **Services**:

A diagram of a software application

Description automatically generated

* + **User Service**: Handles user profiles, authentication, and updates.
  + **Product Service**: Manages product catalogue, search, and product pages.
  + **Cart Service**: Stores and retrieves items in the user's cart.
  + **Payment Service**: Processes payments without needing product or user data.
  + **Order Service**: Tracks orders, optimizes warehouse selection, and manages delivery routes.
  + **Invoice Service**: Generates invoices post-delivery.
  + **Review Service**: Handles product reviews.

1. **Data Isolation**:
   * Each service maintains its own database (e.g., user data in one DB, product data in another).
2. **Communication**:
   * **Direct Calls (Synchronous)**: Cart service calls payment service for transaction completion.
   * **Event-Driven (Asynchronous)**: Order service notifies invoice service upon order completion.
3. **Deployment**:
   * Each microservice runs on independent app servers, facilitating horizontal scaling and fault isolation.

**Advantages of Microservices:**

* **Scalability**: Services can be scaled independently (e.g., more instances of product service during sales).
* **Fault Tolerance**: Failure in one service (e.g., payment) doesn’t crash the entire system.
* **Development Speed**: Teams can work on services in parallel using different tech stacks.
* **Separation of Concerns**: Clear boundaries and ownership between services.

**Best Practices:**

1. **Avoid Over-Splitting**:
   * Ensure logical separation. For example:
     + Good: Authentication and user profile are one service.
     + Bad: Login/logout split from user profile.
2. **Eventual Consistency**:
   * Use tools like distributed transactions, event sourcing, or CQRS to address data inconsistency.
3. **Observability**:
   * Use monitoring and logging systems (e.g., Prometheus, Grafana) for debugging and system health.
4. **APIs**:
   * Design robust APIs with versioning for seamless service communication.
5. **CI/CD Pipelines**:
   * Automate build, test, and deployment processes for each service.

## API Gateway, Load Balancers, and Microservices

**Introduction to Rate Limiting**

* **Rate Limiting**:
  + Applies restrictions on the number of requests within a given time frame.
  + Two common approaches:
    - At **API Gateway**: Ideal for general rules like limiting requests per IP.
    - At **Service Level**: Suitable for specific business logic like limiting profile page loads per user.

**What is an API Gateway?**

* **Definition**: A specialized load balancer aware of microservices.
* **Key Features**:
  + Maps endpoints to corresponding microservices.
  + Common functionalities like **rate limiting** and **authentication** are built-in.
  + Acts as a **router** for requests.
* **Examples of API Gateway Usage**:
  + /product/iphone → Product microservice.
  + /profile/anshuman → User service.
* **Implementation**:
  + Configuration involves creating a mapping (e.g., a hash map) of endpoints to services.

**Comparison: API Gateway vs Load Balancer vs Reverse Proxy**

* **Load Balancer**:
  + Distributes incoming traffic across multiple machines.
  + Essential in monolithic architecture for managing load effectively.
* **API Gateway**:
  + Similar to a load balancer but application-aware.
  + Handles routing, rate limiting, and other logic.
  + Often deployed behind a load balancer.
* **Reverse Proxy**:
  + Relays requests between clients and servers.
  + API Gateway can be seen as a more advanced reverse proxy.

**How Requests Flow:**

1. **Request comes to the Load Balancer**:
   * Balances traffic among available API Gateway instances.
   * Ensures scalability by adding/removing API Gateway machines.
2. **API Gateway Processes the Request**:
   * Matches the endpoint with a microservice.
   * Redirects the request to the appropriate service cluster.
3. **Service Cluster**:
   * Each microservice typically has its own load balancer.
   * Distributes traffic among application servers within the service.

**Microservices and Endpoints:**

* **Endpoints**:
  + Every URL is an endpoint (e.g., /profile/anshuman).
  + Endpoints serve **UI content** or trigger API calls.
  + Examples:
    - Pre-filled UI data (static or dynamic).
    - Asynchronous/synchronous API calls.
* **Microservices**:
  + Serve specific functionalities or data requested via endpoints.
  + Organized as clusters with load balancers.

**Cluster Setup in Microservices:**

* **Cluster**:
  + A collection of app servers for a single microservice.
* **Load Balancer**:
  + Decides which app server to handle the request.
  + Ensures optimal distribution of requests within the cluster.

**Key Takeaways:**

* **API Gateway** simplifies the management of microservices by centralizing routing, rate limiting, and authentication.
* **Load Balancers** remain crucial at different levels:
  + For API Gateway machines.
  + Within microservice clusters.
* **UI and Microservices** are tightly integrated:
  + UI endpoints either serve static content or interact with microservices.
  + Microservices operate as backend functionalities exposed via APIs.

## Monoliths, Microservices, and API Gateway

1. **Typical Microservices-Based UI Architecture:**

* Example: Facebook Web Page
  + The UI has distinct sections like **Menu**, **News Feed**, **Ads**, **Events**, and **People You May Know (PYMK)**.
  + Each section can be managed by separate microservices.
  + Structure is sent by one microservice, and content is loaded asynchronously through individual API calls.

1. **Advantages of This Architecture:**
2. **Faster Page Load:**
   * The page structure (HTML) is fetched quickly.
   * Content is loaded asynchronously, allowing partial interactions while other data loads.
3. **Decoupling:**
   * Each feature or section is handled by its own microservice.
   * Easier to modify or update specific components without impacting others.
4. **Lazy Loading:**
   * Only fetch data as needed, e.g., when scrolling down a feed or interacting with specific UI sections.
5. **Workflow in Microservices-Based UI:**

* **Structure Microservice**:
  + Provides the overall layout and placeholders for the UI.
* **Content Microservices**:
  + Fetch content for specific sections asynchronously.
  + Examples:
    - News Feed: /newsfeed?user\_id=...
    - Ads: /ads?user\_id=...
    - Events: /events?user\_id=...
    - PYMK: /pymk?user\_id=...

1. **API Gateway:**

* **Role**:
  + Acts as an entry point for all client requests.
  + Forwards requests to respective microservices or their load balancers.
* **Use Case**:
  + Simplifies client-server communication by abstracting the microservice infrastructure.

1. **Key Concepts:**

* **Monolith in Context**:
  + Even if the backend is microservices-based, a monolithic UI combines all front-end logic in a single application.
* **Microservices**:
  + Decoupled, independent services managing specific functionalities (e.g., News Feed, Ads).
  + Prioritize separation of databases and reduce shared dependencies.
* **Service-Oriented Architecture (SOA) vs. Microservices**:
  + SOA allows shared databases and synchronous communication.
  + Microservices emphasize independence, asynchronous communication, and eventual consistency.

1. **Misconceptions and Best Practices:**

* **UI Module Division ≠ Microservices**:
  + Breaking UI into modules (e.g., Angular/React) is not microservices unless hosted independently with distinct codebases.
* **Overloading Microservices**:
  + Avoid creating overly granular microservices. Services should be "meaty" enough to justify their independence while avoiding inefficiencies.

**Key Takeaways:**

* Microservices enable faster UI loading, flexibility, and scalability.
* The API Gateway plays a central role in managing client requests and routing them efficiently.
* Proper design of microservices architecture avoids pitfalls like over-coupling or excessive granularity.

## Microservices vs. Monolith Overview:

1. **Microservices**:
   * A microservice is a modular, independent service that can be developed and deployed individually.
   * **Separation of Concerns**: Each microservice is responsible for a specific set of functionalities.
     + For example, in a system with products and users, there can be a **Products Microservice** and a **Users Microservice**.
   * **Independent Tech Stacks**: Microservices can be developed using different technologies, frameworks, and databases suited for each use case.
2. **Monolith**:
   * A monolithic application is a single unified codebase where all the functionalities are tightly coupled.
   * **Unified Tech Stack**: In a monolithic system, the entire application typically uses the same programming language, frameworks, and database.

**Key Characteristics of Microservices:**

1. **Independent Microservices**:
   * Microservices can use different programming languages, databases, and other technologies based on the specific requirements of each service.
     + **Example**:
       - *Products Microservice*: Can be written in Java and use MongoDB for data storage and Redis for caching.
       - *Users Microservice*: Can be written in Python and use MySQL for data storage, possibly with replicas for caching.
   * A diagram of a computer service

     Description automatically generatedThis independence provides flexibility to choose the best technologies suited for each microservice's task.
2. **Separation of Responsibility**:
   * Each microservice is responsible for a specific functionality or set of functionalities, making it easier to manage and maintain.
     + **Example**:
       - *User Microservice*: Manages user profiles, authentication, and user data like addresses, usernames, and passwords.
       - *Product Microservice*: Manages information about products such as categories, prices, etc.
   * This clear division ensures that developers can focus on specific microservices without worrying about the rest of the system.
3. **Tech Stack Independence**:
   * With microservices, each service can choose its own tech stack, databases, and frameworks, based on its use case.
     + For example, one service might use a **NoSQL** database (e.g., MongoDB) due to its flexible schema requirements, while another service might prefer a **SQL** database (e.g., MySQL) due to its structured and normalized data needs.
   * This flexibility allows the system to be optimized for each microservice’s unique needs.

**Advantages of Microservices:**

1. **Customization**:
   * Microservices allow you to make the best possible choices for each service based on its needs without being constrained by the overall system.
     + For example, choosing a database based on whether the system is read-heavy or write-heavy.
2. **Flexibility in Technology Stack**:
   * The ability to choose the best technology (e.g., Java, Python, Ruby, C++) for each microservice leads to more efficient development.
   * Different microservices can use completely different technologies, which helps address specific needs more effectively.

**Key Insights:**

* **Decoupling**: The microservices approach allows developers to break down large systems into smaller, manageable parts. Each microservice works independently, which reduces the complexity of managing the overall system.
* **Autonomy**: The freedom to choose the technology stack and database for each microservice allows for more tailored solutions that better meet the needs of specific services.

## Pros for Microservices

1. **Separation of Concerns in Microservices**:
   * **Tech Stack Independence**: Each microservice can have its own tech stack, programming language, and database suited to its specific use case. For example:
     + **Product Microservice**: Built in Java, using MongoDB for storage and Redis for caching.
     + **User Microservice**: Built in Python, using MySQL for storage and a replica for caching.
   * **Decoupling**: Each microservice can function independently, and developers can choose the optimal technology for the task at hand without worrying about other services.
2. **Independent Deployment**:
   * **Deployability**: Changes made to one microservice (e.g., the product microservice) do not require deployment of other services (e.g., the user microservice). Only the microservice that undergoes changes needs to be redeployed.
   * **Agility in Deployment**: This makes the deployment process quicker and more isolated, reducing the risk of downtime in unrelated services.
3. **Independent Testing**:
   * **Smaller Scope**: With clear responsibility for each microservice (e.g., the product microservice handles only product-related logic), testing is focused and more manageable. There's no need to test the entire system when changes are made to a single microservice.
   * **Simpler Testing**: Developers only test the functionality related to the microservice they are working on.
4. **Independent Scaling**:
   * **Traffic Handling**: Microservices can scale independently based on traffic demands. For instance:
     + The product page may require more servers to handle high traffic (e.g., 100 million or even a billion requests per day).
     + The user service might require fewer resources because it's not as heavily trafficked (e.g., only 1 million requests per day).
   * **Flexible Resource Allocation**: This allows for more efficient use of infrastructure resources and cost optimization.
5. **Robustness and Fault Tolerance**:
   * **Fault Isolation**: Even if one microservice fails (e.g., the user service), other services (e.g., the product service) can continue to function. This is crucial for maintaining system availability and minimizing the impact of failures.
   * **Limited Impact of Failures**: In contrast to monolithic systems, where a failure in one part of the system can bring down the entire application, microservices ensure that issues are isolated to the affected service.
6. **Personalization of Product Information**:
   * **Separation of Concerns with Personalization**:
     + The product service doesn't directly need to know about the user service or user IDs to perform tasks like showing personalized product recommendations.
     + Personalization can be achieved by passing user attributes (such as location, interests, and device type) as parameters to the product service's functions.
     + The user service can maintain the list of user attributes, and other services (such as a frontend) can fetch these attributes and call the product service with the necessary context.

**Summary of Pros for Microservices:**

* **Separation of concerns** enables:
  + Independent tech stacks, databases, and deployment.
  + Simplified testing, as each service has a well-defined scope.
  + Independent scaling based on traffic patterns and resource requirements.
  + Improved fault tolerance and robustness, as issues in one service don't affect the entire system.

## Cons for Microservices

**What Are Microservices?**

* Microservices are an architectural style that structures an application as a collection of small, independent, and loosely coupled services.
* Each microservice focuses on a specific functionality and communicates through APIs.

A diagram of a software process

Description automatically generated

**Role of an API Gateway**

* Serves as an entry point for clients (e.g., mobile apps, browsers).
* Manages authentication, rate limiting, encryption/decryption, and request routing.
* Simplifies communication between clients and microservices.

**Diagram Context**

* The diagram showcases a microservices architecture:
  + **Mobile/Browser** clients send requests.
  + Requests go through:
    1. A Load Balancer.
    2. API Gateways for routing and management.
    3. Microservices (e.g., Product Service, Order Service).
  + **Product and Order services have their own load balancers**, app servers, databases, and caches.

**Advantages of Microservices**

1. **Scalability:**
   * Services can be scaled independently based on demand.
2. **Fault Isolation:**
   * Failure in one microservice doesn't affect others.
3. **Flexibility:**
   * Services can be written in different programming languages.
4. **Faster Deployment:**
   * Individual services can be updated and deployed independently.

**Disadvantages of Microservices**

1. **Increased Latency:**
   * Additional layers (e.g., API Gateway, inter-service communication) introduce more network calls, resulting in higher latency.
   * In the monolithic architecture, fewer layers exist: requests directly interact with app servers and databases.
   * In microservices:
     + **Requests might need to interact with multiple services (each with its load balancer, app server, and database) to fulfil a single client request**.
     + More network calls mean more delays compared to a monolithic system.
2. **Complexity:**
   * More services lead to increased operational complexity.
   * Coordination between multiple services can make feature development challenging.

**Example of Over-Engineering:**

* Over-segmenting functionalities into numerous microservices can:
  + Increase latency.
  + Make feature development and debugging harder due to excessive interdependencies.

1. **Higher Costs**
   * **Reason:**
     + Microservices require more infrastructure compared to monoliths.
     + Each microservice needs:
       - At least one **API Gateway**.
       - Its own **app server**.
       - Its own **database** or **cache**.
       - Its own **load balancer**.
     + This applies regardless of traffic or the size of the application.
   * **Comparison with Monoliths:**
     + Monoliths require fewer resources:
       - One or two app servers.
       - A single database.
   * **Impact of Virtualization/Docker:**
     + While tools like Docker reduce overhead, they still introduce their own resource usage.
     + Additional layers like API Gateway and load balancer further increase costs.
2. **Increased Complexity**
   * Microservices require more coordination:
     + **API Gateway and Load Balancer:** These components don’t exist in monoliths and add extra layers of processing.
     + **Individual Service Work:** Each app server in a microservice architecture performs only a small portion of the overall workload, leading to inefficiencies.
   * Complexity makes it harder to maintain and debug systems.
3. **Logging and Tracing Are Harder**
   * **Monolithic Logging:**
     + Logs for all functionalities are centralized in a single location.
     + Easier to filter logs and debug issues.
   * **Microservices Logging:**
     + Requests often span multiple services.
     + Debugging requires checking logs across multiple services, making the process more time-consuming.
   * **Observability:**
     + Observability tools like **ELK Stack**, **Splunk**, and **Grafana** are essential for managing distributed systems.
     + Observability ensures centralized monitoring, logging, and tracing across services.
4. **Data Inconsistency**
   * **Reason:**
     + Microservices may maintain duplicate data across services.
     + Example: An **Order Service** and a **Payment Service** both track whether a payment is completed.
     + No guarantee exists that data will remain synchronized across services.
   * **Challenges:**
     + Data inconsistency can lead to logical errors and system failures.
   * **Solutions:**
     + Design patterns like distributed transactions or eventual consistency mechanisms help address this challenge.
5. **Performance Overheads**
   * More services introduce additional **network latency** due to:
     + API Gateway processing.
     + Inter-service communication.
     + Load balancer delays.
   * Requests often require interaction with multiple services, leading to slower response times.

**When to Choose Microservices?**

* Avoid microservices in early-stage projects or for small applications due to:
  + High setup and maintenance costs.
  + Complexity in managing distributed systems.
* Opt for microservices only when:
  + The project grows too large to manage as a monolith.
  + Teams require independent development and deployment cycles for different functionalities.

**Key Problems to Solve in Microservices**

1. **Observability:**
   * How to monitor, log, and trace requests across distributed services effectively.
   * Tools: ELK Stack, Splunk, Grafana, Jaeger.
2. **Data Consistency:**
   * How to maintain consistency in distributed systems when data is duplicated across services.
   * Techniques:
     + Distributed transactions.
     + Eventual consistency.
3. **Cost Management:**
   * While costs cannot be eliminated, optimization techniques like asynchronous processing and prioritizing synchronous requests can reduce overhead.

**Conclusion**

* Microservices are not a one-size-fits-all solution.
* Their advantages include scalability, fault isolation, and independent deployments.
* However, they come with significant drawbacks such as higher costs, complexity, and challenges in logging, tracing, and maintaining data consistency.
* Thoughtful evaluation is necessary before adopting microservices to ensure they align with the project’s needs.

## Microservices Some Topics

**1. Microservices Overview**

* Microservices architecture involves splitting applications into smaller, independent services.
* Key focus areas in interviews:
  + **What are Microservices?**
    - Definition, examples (e.g., Flipkart use case).
  + **When to use Microservices?**
    - Use-case scenarios and justification.
  + **Examples of Microservices:**
    - Services like user authentication, inventory, payment gateway, etc.

**2. Common Interview Topics on Microservices**

* **Distributed Transactions in Microservices:**
  + Addressed using patterns like **SAGA** and **CQRS**.
* **Observability in Microservices:**
  + Focus on logging, tracing, and monitoring across services.
* **Minimizing Impact of Service Failure:**
  + Use **Circuit Breaker Pattern** to reduce dependency impact.
* **Service-to-Service Communication:**
  + Discuss how one service failing (e.g., Service 2) affects others (e.g., Service 1 and 3).
  + Strategies to avoid cascading failures.

**3. Challenges in Microservices**

1. **More Layers, More Latency:**
   * Introduced due to distributed architecture.
   * Solution: **Event-Driven Architecture** for faster communication.
2. **Observability Challenges:**
   * Logging, tracing, and debugging become complex with multiple services.
   * Solution: Use **centralized logging and tracing tools**.
3. **Data Consistency Issues:**
   * Maintaining consistency across distributed services.
   * Solutions:
     + **SAGA Pattern**: Manages transactions across services.
     + **CQRS Pattern**: Separates command (write) and query (read) models for better data handling.
4. **Handling Service Failures Gracefully:**
   * Example: Service 1 dependent on Service 2; if Service 2 fails, Service 1 can also get overwhelmed.
   * Solution: Use **Circuit Breaker Pattern** to:
     + Detect failures early.
     + Stop cascading failures by cutting off repeated failed requests.
5. **External Service Dependencies:**
   * Services like payment or notification rely on external systems.
   * Design strategies to reduce dependency on third-party service failures.
6. **Authentication Challenges:**
   * API Gateway authentication is critical for secure communication.
   * Solutions include token-based authentication and rate limiting.

**4. Key Patterns and Techniques to Learn**

1. **Circuit Breaker Pattern:**
   * Detects and handles service failure gracefully.
   * Prevents a service from being overwhelmed with requests when another service fails.
2. **SAGA Pattern:**
   * Manages distributed transactions by breaking them into smaller, manageable steps.
   * Example: Payment service ensuring funds are deducted only if the order is successful.
3. **CQRS Pattern:**
   * Separates write and read operations for scalability and better performance.
4. **Event-Driven Architecture:**
   * Reduces latency by facilitating asynchronous communication between services.

**5. Agenda for Remaining Classes**

1. **Reducing Latency:**
   * Study **Event-Driven Architecture**.
2. **Observability:**
   * Learn techniques for **logging, tracing, and monitoring** in microservices.
3. **Distributed Transactions:**
   * Understand **SAGA** and **CQRS** patterns.
4. **Graceful Service Failure Handling:**
   * Implement the **Circuit Breaker Pattern**.
5. **Design for External Dependencies:**
   * Strategies for handling services like payments or notifications that rely on external systems.
6. **Authentication in API Gateway:**
   * Secure service-to-service communication using proper authentication mechanisms.

## Observability

Observability is a concept in software engineering that revolves around the ability to understand, monitor, and debug a system effectively. It's particularly crucial in modern, complex architectures like microservices. Here's a summary of the key ideas:

**What is Observability?**

Observability refers to:

1. **Detection**: Identifying when something goes wrong in your system. This could involve setting up alarms, notifications, or other mechanisms to detect errors, performance degradation, or failures.
2. **Diagnosis**: Analysing logs, metrics, or traces to determine the root cause of an issue and identify where the problem occurred.
3. **Validation**: After fixing the issue, confirming through logs, metrics, and system behaviour that the problem has indeed been resolved.

**Importance of Observability**

* Modern systems are complex and may fail due to a variety of reasons, such as:
  + Bugs in the code
  + Hardware or network failures
  + Unexpected user inputs or corner cases
* Observability ensures that you can quickly detect and respond to these failures, minimizing downtime and improving reliability.

**Logs, Metrics, and Tracing**

1. **Logs**: Records of events or errors that occur in the system. They help in understanding what happened at each step.
2. **Metrics**: Quantifiable data points like response times, memory usage, CPU utilization, or error rates. Metrics help assess the health and performance of the system.
3. **Tracing**: Follows a request as it traverses different components or services in a system, identifying bottlenecks or points of failure.

**Tracing Example**

A diagram of a computer network

Description automatically generated

* In a microservices architecture (e.g., Messenger app):
  + A message goes through various components: API Gateway → Message Service → Database → Queue → Notification Service.
  + Tracing involves assigning a **common ID** (e.g., a message ID) to this request and logging it at every step.
  + If the receiver doesn't get a notification, tracing helps identify which step in the pipeline failed.

**Centralized Logging**

* Logs from all components need to be aggregated in a single place for effective analysis.
* Tools like **Elasticsearch**, **Kibana**, and **Logstash** (the ELK stack) can centralize and visualize logs.
* Paid tools like **Splunk** also provide similar functionality.

**Observability in Monoliths vs. Microservices**

* In monolithic systems, observability is simpler since all components are in one place.
* In microservices, it becomes more complex due to:
  + Multiple independent services
  + Various communication protocols (e.g., HTTP, message queues)
  + Diverse tech stacks

**Challenges and Solutions**

* **Challenge**: Asynchronous communication can fail (e.g., messages may not reach a queue).
* **Solution**: Implement retry mechanisms, use robust queues (like Kafka), and ensure that each step logs its status with a traceable ID.

**Practical Tools**

* **Splunk**: A paid solution for log management.
* **ELK Stack**: Free and open-source alternative for logging and visualization.
* **Sentry**: Focused on error detection and monitoring.
* **Distributed Tracing Frameworks**: Tools like **Jaeger** or **OpenTelemetry** support tracing across services.

**1. Distributed Tracing**

* **Key Insight**: Distributed tracing involves tracking the flow of a request across multiple microservices or systems. A **common trace ID** is essential to correlate logs across services.
* **How It Works**:
  + Each service in the chain logs messages with a trace ID attached. This trace ID remains constant throughout the request's lifecycle.
  + Logs from all services are collected in a centralized system like **ELK**, **Splunk**, **Datadog**, or others.
  + The trace ID helps pinpoint where a failure occurred or where a request might have stalled.
* **Free and Paid Options**:
  + Free: Open-source tools like **ELK (Elasticsearch, Logstash, Kibana)**.
  + Paid: **Datadog**, **Stackdriver**, **New Relic**, etc.
* **Limitations**: Some solutions might not support full-text search, restricting you to search by trace ID or specific fields.

**2. Metrics Monitoring**

* **Key Insight**: Observing metrics like CPU usage, memory usage, request latency, and database response time helps track system health and detect bottlenecks.
* **How It Works**:
  + Metrics are periodically published to a **time series database (TSDB)**.
  + These metrics can be visualized using tools like **Prometheus + Grafana**, **Datadog**, or **New Relic**.
  + TSDB stores values for specific time intervals (e.g., one-minute averages) and allows aggregation over time (e.g., min, max, average).
* **Example Use Case**:
  + Plot CPU usage for the past 24 hours.
  + Raise an alert if CPU usage exceeds 90% for 5 consecutive minutes.

**3. Alarms**

* **Key Insight**: Alerts notify you of anomalies like high latency or CPU usage exceeding a threshold.
* **How It Works**:
  + Set rules on your monitoring system to trigger alerts when metrics cross defined thresholds.
  + Integrate with tools like **PagerDuty** for phone call alerts or Slack/email for notifications.
* **Example**:
  + If database response time exceeds 500ms, trigger an alarm and notify the on-call engineer via PagerDuty.

**4. Combining Tracing, Metrics, and Alarms**

* Distributed tracing helps identify **where** an issue occurred.
* Metrics and dashboards monitor **what** is happening in the system.
* Alerts notify you **when** something goes wrong.
* Together, these provide comprehensive **observability**.

**Interview Insight**

When asked about monitoring failures in a microservices architecture:

* Highlight **distributed tracing** as the method to track requests across services.
* Emphasize the importance of a **centralized logging system** to collect and analyze logs.
* Discuss using **time-series metrics** to monitor system performance and setting up alerts for anomalies.

## Time Series Databases and Metrics Analysis

**1. Introduction to Metrics and Time Series Database (TSDB):**

* A **metric** refers to a measurement that provides insight into system performance (e.g., latency, error rate).
* A **time series database** stores time-stamped data associated with specific metrics.
  + Example: Key (latency), Time-Stamp (11:01 PM), Value (200ms).

**2. Importance of Metrics for Debugging:**

* Metrics help diagnose performance issues without searching logs manually.
  + Example: If users complain that a website was slow at 11:30 PM, analysing metrics like CPU usage, memory usage, and latency on dashboards can quickly identify the issue.

**3. Key Features of a TSDB:**

* **Data Organization:**
  + Data is stored as a key (metric) associated with time-stamp-value pairs.
  + Example: latency -> [ (11:01 PM, 200ms), (11:02 PM, 400ms), ... ].
* **Granularity Flexibility:**
  + TSDB should allow users to view data at different levels of granularity:
    - Per minute
    - Per 5 minutes
    - Per hour (e.g., for one-month data visualization).
* **Aggregation Support:**
  + TSDBs should efficiently aggregate data into buckets (e.g., hourly buckets) using operations like:
    - **Sum**
    - **Average**
    - **Maximum**

**4. Aggregation Process:**

* Aggregation simplifies visualizing large datasets over time.
  + Example: For hourly data, aggregate minute-level values into hourly buckets.
  + Aggregation operations can also be specified by users depending on their needs.

**5. Why Aggregation is Critical:**

* Data for each time point often involves aggregation across multiple machines before storage.
  + Example: Compute the average latency across machines to get a single value for 11:01 PM.

**6. Popular Time Series Databases:**

* Common TSDBs include:
  + **InfluxDB**
  + **TimescaleDB**
  + **Prometheus**
  + **Amazon Timestream**
  + **Apache Pinot**
  + **Apache Druid**
  + **Graphite**
  + **Redis TimeSeries**
* Tools with built-in TSDBs:
  + **New Relic**
  + **Datadog**
  + **Dynatrace**
  + These services are managed and charge for usage.

**7. Designing Your Own TSDB:**

* **Core Operations:**
  + Store key -> time-stamp-value pairs.
  + Perform quick aggregations (sum, average, max).
* Designing TSDB is a data structure and algorithm (DSA) problem.
  + Challenges:
    - Efficient storage of time-stamped data.
    - Fast computation of aggregates for specified time buckets.

**8. Example Question:**

* How would you design a TSDB that supports storing time-stamp-value pairs and performs efficient aggregations?
  + This is a classic **DSA problem** that involves:
    - Efficient indexing for time-stamped data.
    - Optimized algorithms for aggregation queries.

## User Engagement Metrics and Their Tracking

**Approaches to Track Metrics**

1. **Logging to a Time Series Database (TSDB):**
   * A TSDB is a specialized database optimized for handling time-stamped data.
   * Process:
     + Assign a key (e.g., "live users") and a value (e.g., 100).
     + Use a library function (e.g., ODS.publish(key, value)).
     + The TSDB automatically timestamps the key-value pair and logs it.
     + Over time, the data syncs with the central TSDB, enabling easy access to metrics.
   * Example:
     + Key: "latency"
     + Value: Response time for an operation.
2. **Logging in Files (Centralized Search and Aggregation):**
   * If the data is logged in files (instead of a TSDB), the following can be done:
     + **Centralized Log Management**:
       - Use tools like the ELK Stack (Elasticsearch, Logstash, Kibana) to centralize logs.
     + **Search and Aggregation**:
       - Search for specific logs using matching patterns.
       - Aggregate the values (e.g., summing up or averaging) to calculate metrics like the number of live users over the last 24 hours.
3. **Storing Metrics in a Distributed Database:**
   * Metrics may be saved across multiple machines or database shards.
   * Retrieving metrics in such cases involves:
     + **MapReduce Technique**:
       - **Map**: Fetch relevant data from all machines.
       - **Reduce**: Aggregate the fetched data to calculate the final metric.
     + Suitable for distributed systems like HBase, Hadoop Distributed File System (HDFS), or Hive.

**Key Points to Remember**

* **Time Series Database**: Best for real-time, continuous metric tracking (e.g., number of live users, latency).
* **Log Search and Aggregation**: Useful when metrics are stored in logs; requires centralized log management tools.
* **Distributed Databases and MapReduce**: Necessary when metrics are spread across multiple systems and need aggregation.

**Tool Examples**

* **Time Series DBs**: Prometheus, InfluxDB.
* **Log Management Tools**: ELK Stack (Elasticsearch, Logstash, Kibana).
* **Distributed Data Processing**: Hadoop, Hive, HBase.