SYSTEM Design

Designing a high-throughput system capable of **handling millions of real-time requests** for services like Pricing, Search, Booking, and Reviews requires a robust architecture that emphasises speed, scalability, reliability, flexibility, observability, security and robust and sustainable Testing (**SSREFOST**). Below is a structured approach to this system design, including high-level and low-level architectural considerations, technology choices, and best practices.

The provided document outlines the architecture design, components, technologies, and strategies for building a high-volume throughput, real-time data system. It ensures scalability, reliability, flexibility, observability, and security while addressing challenges like **schema evolution** and **fault tolerance** and also handling millions of requests with real-time data processing capabilities..

1. Architectural Design Patterns for Consideration

Frontend Architectures:

- **Monolithic Architecture:** A single unified codebase. Simpler but less scalable and flexible.
- **Micro-Frontend Architecture:** Decomposes the frontend into smaller, manageable pieces corresponding to different services. Enhances scalability and independent deployment.

Backend Architectures:

- **Monolithic Architecture:** All backend services are part of a single application. Easier to develop initially but challenging to scale and maintain.
- **Microservices Architecture:** Each service (Pricing, Search, Booking, Reviews) is developed, deployed, and scaled independently. Offers better scalability and fault isolation.
- **Event-Driven Architecture:** Services communicate via events, promoting loose coupling and real-time processing. Suitable for real-time data systems.
- **SOA**

Backend

2. High-Level Architecture

The high-level architecture integrates various components to ensure seamless interaction between users and services.

Components:

- **API Gateway:** Serves as a **single entry point** for all client requests, routing them to appropriate services. Handles concerns like rate limiting, authentication, and request logging.

- **Vertical Services:**
  - **Pricing Service:** Manages pricing information and calculations.
  - **Search Service:** Handles search queries and retrieval of relevant data.
  - **Booking Service:** Manages reservations and booking transactions.
  - **Reviews Service:** Handles user reviews and ratings.
- **Service Communication:** Facilitated through synchronous (REST/gRPC) and asynchronous (message queues) methods.

- **Data Layer:** Comprises databases and data storage solutions for each service, ensuring data integrity and availability.

- **Configuration Management:** Centralised management of service configurations to maintain consistency across environments.

- **Logging and Monitoring:** Implements observability tools to monitor system health, **and** performance and detect anomalies.

- **Authentication and Authorization:** Ensures secure access to services and data, managing user identities and permissions.

- **Error Handling:** Standardized mechanisms to handle exceptions and failures gracefully, ensuring system resilience.

- **CI/CD Pipeline:** Automates the build, test, and deployment processes to ensure rapid and reliable releases.

- **Containerisation and Orchestration:** Uses containers to package services and orchestration tools to manage deployment, scaling, and operations.

- **Reporting Service:** Aggregates logs and metrics for analysis, using tools like Prometheus and ELK stack for monitoring and alerting.

High-Level Architecture Diagram:

```
+-------------------+

|                   |

|   API Gateway    |

|                   |

+--------+----------+

         |

+--------+----------+

|             |

|  Service Mesh    |

|             |

+--------+----------+

         |

+--------+----------+

|                   |

|  Vertical Services|

|                   |

+--------+----------+

      |

+--------+----------+

|                   |

|   Data Layer    |

|                   |

+-------------------+
```

3. Low-Level Architecture

The low-level architecture delves into the specifics of each component, detailing interactions, data flow, and technology choices.

API Gateway:

- **Responsibilities:**
    - Routing requests to appropriate services.
    - Load balancing incoming traffic.
    - Implementing security measures like SSL termination and request authentication.
    - Rate limiting to prevent abuse.
- **Technology Choices:**
    - **Kong:** An open-source API gateway with a rich plugin ecosystem.
    - **NGINX:** High-performance web server and reverse proxy.
    - **AWS API Gateway:** Managed service for creating and managing APIs.

Vertical Services:

- **Design:**
    - Each service is a microservice with its own codebase, database, and deployment pipeline.
    - Services communicate over well-defined interfaces (APIs).
- **Technology Choices:**
    - **Programming Languages:**
        - **Node.js:** Suitable for I/O-bound services like Search.
        - **Python:** Ideal for data-intensive services like Pricing.
        - **Java:** Preferred for transaction-heavy services like Booking.
    - **Frameworks:**
        - **Express.js** for Node.js.
        - **Spring Boot** for Java.
        - **Django** for Python.

Service Communication:

- **Synchronous Communication:**
    - **REST:** Simple and widely adopted.
    - **gRPC:** High-performance, suitable for internal service communication.
- **Asynchronous Communication:**
    - **Message Queues:**
        - **RabbitMQ:** Reliable message broker.
        - **Apache Kafka:** Suitable for high-throughput, real-time data streaming.

Data Layer:

Databases:

- **Relational Databases:**
  - **PostgreSQL:** For transactional data like bookings and pricing that need ACID compliance.
  - **MySQL:** Optional for horizontal scalability with clustering (e.g., Percona or Galera Cluster).
- **NoSQL Databases:**
  - **MongoDB:** For schema-less data, e.g., user-generated reviews and logs.
  - **Cassandra:** High availability and linear scalability for distributed data storage.
- **Search Engine Database:**
  - **Elasticsearch:** Optimized for text search queries used in the Search service.
- **Caching Layer:**
  - **Redis:** Fast in-memory key-value storage for caching frequently accessed data.
  - **Memcached:** Lightweight and optimized for fast lookups.
- **Message Queues:**
  - **Apache Kafka:** Distributed streaming for event-driven systems, ensuring reliable and scalable communication.
  - **RabbitMQ:** For lightweight messaging in services requiring ordered processing.

Configuration Management:

- **Consul or etcd:** Centralized storage for configurations.
- **Kubernetes ConfigMaps and Secrets:** Manage sensitive and non-sensitive configurations in containerized systems.

Logging and Monitoring:

- **Log Aggregation:**
  - **ELK Stack (Elasticsearch, Logstash, Kibana):** Logs collection, storage, and visualization.
  - **Fluentd:** Unified log processing and forwarding.
- **Monitoring Tools:**
  - **Prometheus:** Monitoring metrics and alerts.
  - **Grafana:** Data visualization.
  - **Jaeger/Zipkin:** Distributed tracing for debugging service interactions.

Authentication and Authorization:

- **OAuth2.0 and OpenID Connect:** For user identity management.
- **Keycloak/Okta:** Identity provider and Single Sign-On solutions.
- **JSON Web Tokens (JWT):** Token-based authentication for secure API requests.

Error Handling:

- **Retry Mechanisms:** Implement exponential backoff for transient errors.
- **Dead Letter Queues (DLQ):** Handle unprocessable messages in Kafka or RabbitMQ.
- **Fallback Strategies:** Circuit breaker pattern using tools like Resilience4j or Hystrix.

CI/CD Pipeline:

- **Jenkins/GitLab CI/CD:** Build, test, and deploy pipelines.
- **Docker and Kubernetes:** Containerization and orchestration.
- **ArgoCD or Spinnaker:** Continuous delivery for Kubernetes.

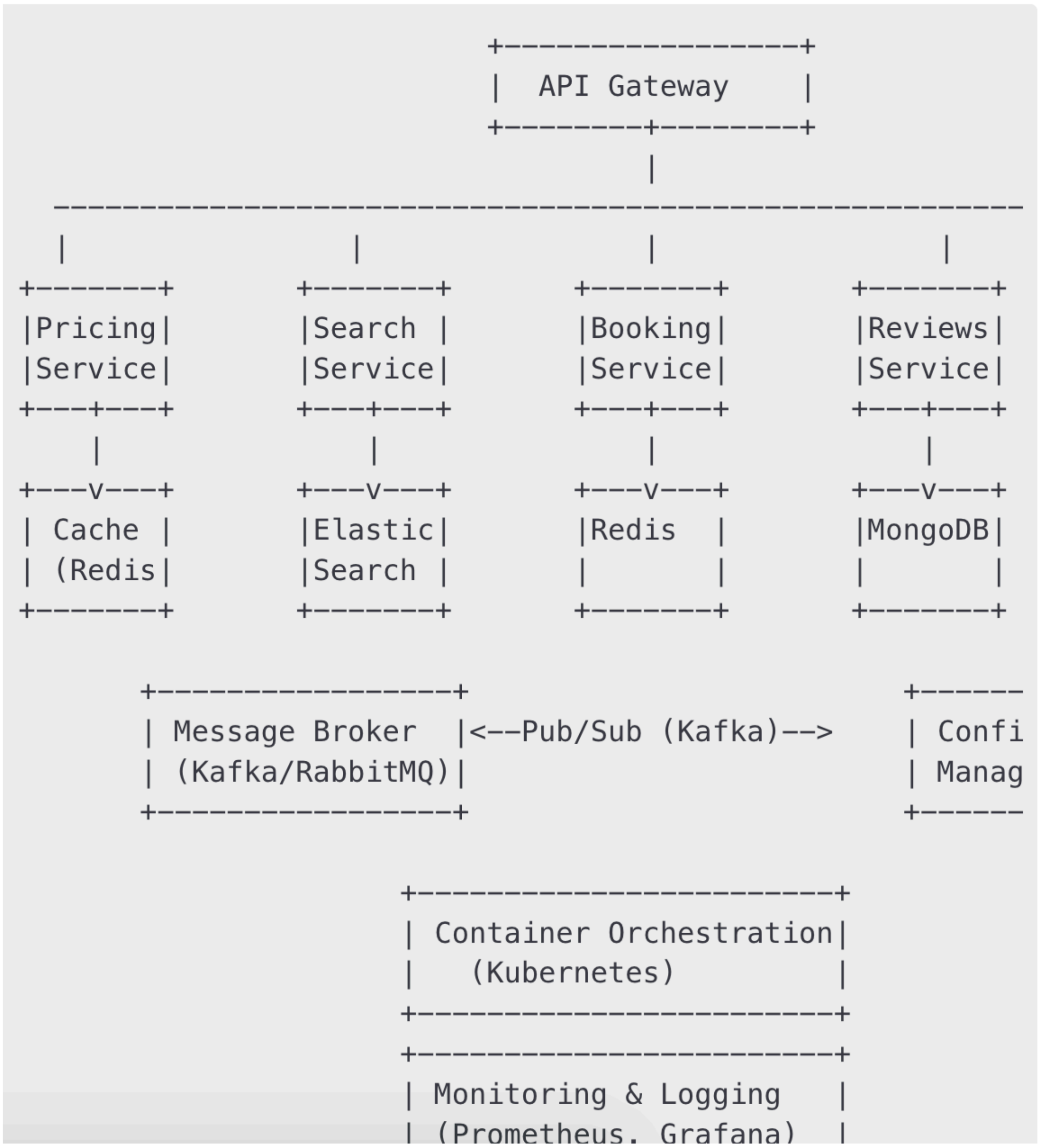
Containerization and Orchestration:

- **Docker:** Lightweight containers for services.
- **Kubernetes:** Manages containerized services, enabling scalability and fault tolerance.
- **Helm Charts:** For defining, installing, and upgrading Kubernetes applications.

Reporting Service:

- **Prometheus:** Metrics collection for observability.
- **Grafana Dashboards:** Real-time visual analytics and alerts.
- **ELK Stack:** Aggregates logs and errors for auditing and compliance.

Low-Level Architecture Diagram



```
                            +------------------+
                            |  API Gateway     |
                            +--------+---------+
                                     |
   --------------------------------------------------------
   |                |                |                |
+-------+       +-------+       +-------+       +-------+
|Pricing|       |Search |       |Booking|       |Reviews|
|Service|       |Service|       |Service|       |Service|
+---+---+       +---+---+       +---+---+       +---+---+
    |               |               |               |
+---v---+       +---v---+       +---v---+       +---v---+
| Cache |       |Elastic|       |Redis  |       |MongoDB|
| (Redis|       |Search |       |       |       |       |
+-------+       +-------+       +-------+       +-------+

       +------------------+                        +------
       | Message Broker   |<--Pub/Sub (Kafka)-->   | Confi
       | (Kafka/RabbitMQ)|                         | Manag
       +------------------+                        +------

                       +-------------------------+
                       | Container Orchestration|
                       |   (Kubernetes)          |
                       +-------------------------+
                       +-------------------------+
                       | Monitoring & Logging    |
                       | (Prometheus, Grafana)   |
```

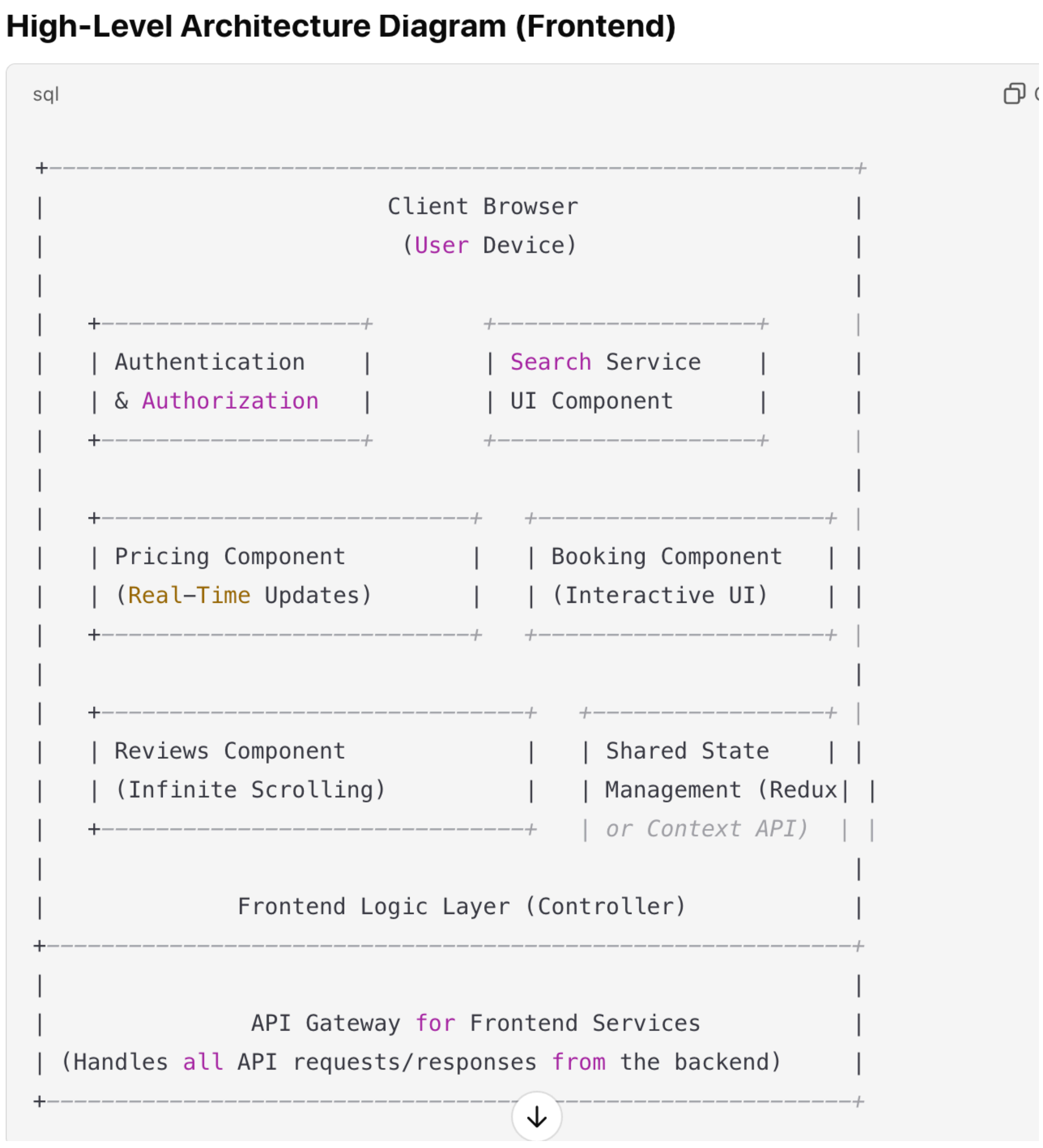
4. Efficiency and Scalability Considerations

1. **Data Structures:**
    - Use **B-Trees** and **Hash Indexes** for database indexing (**B-Trees** are balanced tree data structures used for ordered data storage and efficient range queries, while **Hash Indexes** use hash tables for fast, exact-match lookups but do not support range queries).
    - Utilize **Bloom Filters** in search services for **fast lookups**.
    - Implement **Trie Structures** for **autocomplete** and **search suggestions.**
2. **Size Estimations:**
    - Plan database sharding for horizontal scaling from the start.
    - Estimate **peak loads** using **traffic simulation** tools (e.g., Locust, JMeter).
3. **Production Testing:**
    - Perform **Load Testing** using Apache JMeter and Gatling.
    - Monitor performance metrics and optimize SQL queries.
    - Simulate **Fault Injection Testing** using Chaos Engineering tools (e.g., Gremlin).
4. **Observability Tests:**
    - Validate metrics and logs collection.
    - Implement alerts for anomalies and latency spikes.
5. **Security Tests:**
    - Conduct **Penetration Testing** and **Vulnerability Scans** (SONARQ).
    - Ensure proper encryption (TLS) and secure secrets management.

Front end: Low-level and High-level Architecture

This high-level architecture outlines a scalable, reliable, and flexible frontend system design for high-throughput services such as Pricing, Search, Booking, and Reviews. It incorporates modern architectural practices to handle millions of requests and ensure observability, security, and performance.

This frontend architecture combines modern patterns, technologies, and observability to support scalability, fault tolerance, and performance for high-throughput services. Let me know if you need deeper insights into specific modules or workflows!



High-Level Architecture Diagram (Frontend)

sql

```sql
+------------------------------------------------------------+
|                         Client Browser                     |
|                          (User Device)                     |
|                                                            |
|   +--------------------+        +--------------------+     |
|   | Authentication     |        | Search Service     |     |
|   | & Authorization    |        | UI Component       |     |
|   +--------------------+        +--------------------+     |
|                                                            |
|   +----------------------------+   +---------------------+ |
|   | Pricing Component          |   | Booking Component   | |
|   | (Real-Time Updates)        |   | (Interactive UI)    | |
|   +----------------------------+   +---------------------+ |
|                                                            |
|   +--------------------------------+   +-----------------+ |
|   | Reviews Component              |   | Shared State    | |
|   | (Infinite Scrolling)           |   | Management (Redux| |
|   +--------------------------------+   | or Context API)  | |
|                                                            |
|              Frontend Logic Layer (Controller)             |
+------------------------------------------------------------+
|                                                            |
|               API Gateway for Frontend Services            |
| (Handles all API requests/responses from the backend)      |
+------------------------------------------------------------+
```

1. Key Components in the Frontend High-Level Design:

1. **Client Application (React, Next.js):**
   - CSR (Client-Side Rendering - Loading on demand) and SSR (Server-Side Rendering - Static compiled data) hybrid model for fast loading.
   - Tailwind CSS for styling and responsive design.
2. **API Gateway:**
   - Unified entry point for API calls.
   - Load balancer for distributing traffic.
   - Handles authentication, rate limiting, and CORS.
3. **Service Communication Layer:**
   - GraphQL for flexible query structures.
   - REST API fallback for legacy services.
   - WebSockets for real-time updates (e.g., booking status).
4. **Authentication and Authorization (Auth Service):**
   - OAuth 2.0 (User Identity) and JWT for secure token-based access.
   - Role-based access control (RBAC).

5. **Data Caching and CDN (Content Delivery Network):**
   - Redis or Varnish for caching frequently accessed data (kEY/VALUES).
   - CDN (e.g., Cloudflare) for static asset delivery.
6. **Logging and Monitoring (Observability):**
   - Prometheus for metrics collection.
   - ELK Stack (Elasticsearch, Logstash, Kibana) for centralized logging and visual dashboards.
7. **CI/CD Pipeline:**
   - GitHub Actions / Jenkins for automated builds, tests, and deployments.
   - **Canary** deployments for **rolling out update**s with minimal risk.
8. **Containerisation and Orchestration:**
   - Docker for containerization.
   - Kubernetes for container orchestration, scaling, and failover.

Low-Level Architecture Diagram (Frontend)

```sql
+-----------------------------------------------------------+
|                          CLIENT                           |
|                                                           |
|   +-------------------+   +-------------------+           |
|   | React Components  |   | TailwindCSS for   |           |
|   | (UI Elements)     |   | Styling           |           |
|   +-------------------+   +-------------------+           |
|                                                           |
|   +-----------------------------------------------------+ |
|   | State Management (Redux/Context API)                | |
|   |  - Centralized Store                                | |
|   |  - Actions/Reducers                                 | |
|   +-----------------------------------------------------+ |
|                                                           |
|   +------------------+     +---------------------------+ |
|   | Component Logic  |     | Data Fetching Layer       | |
|   | - Handles events |     | - Axios/Fetch API         | |
|   |   and actions    |     | - Caching via SWR or RTK  | |
|   +------------------+     |                           | |
|                             +---------------------------+ |
+-----------------------------------------------------------+
|                                                           |
|                Communication with Backend                 |
|  - API Gateway for Data Fetching                          |
|  - GraphQL/REST Integration                               |
|  - Error Handling and Retry Mechanism                    |
```

1. Component Breakdown:

UI Layer:

- **Component Library (Storybook):** Standardized UI components.
- **Pages:**
  - Home, Search Results, Booking Flow, Pricing Details, and Reviews.
- **Reusable Components:**
  - Buttons, forms, and modals.

State Management (Redux/RTK Query):

- **Local State:**
  - UI state (e.g., modal visibility).
- **Global State:**
  - Cached API responses and shared data.
- **Middleware:**
  - Redux Thunk or Redux Saga for handling side effects like asynchronous calls.

Controller Layer:

- **API Service Handlers:** Abstracted layer for handling API requests (fetch, axios).
- **Event Handling:**
  - WebSockets for real-time updates (e.g., bookings and reviews).

Data Layer:

- **Caching Strategies:**
    - SWR (stale-while-revalidate) for optimized data fetching.
- **Schema Validation:**
    - JSON Schema or TypeScript interfaces.

2. Dependencies:

1. **API Gateway:**
    - Manages communication between services and frontend.
    - Ensures high availability and load balancing.
2. **Service Communication (GraphQL/REST):**
    - GraphQL for optimized data fetching and flexibility.
    - REST for backward compatibility.
3. **Authentication and Security:**
    - Secure sessions and tokens (JWT).
    - XSS and CSRF prevention.
4. **Error Handling and Monitoring:**
    - Centralized logging via ELK.
    - Frontend crash monitoring (e.g., Sentry).

3. Challenges and Resolutions:

Challenge 1: Scalability

- **Issue:** Handling millions of concurrent requests.
- **Resolution:** Implement CDN, Redis caching, and Kubernetes autoscaling.

Challenge 2: Real-Time Updates

- **Issue:** Booking and pricing require real-time data.
- **Resolution:** Use WebSockets and Event-Driven Architecture with message queues.

Challenge 3: Observability

- **Issue:** Monitoring user interactions and failures.
- **Resolution:** Integrate Prometheus and ELK for real-time metrics and logs.

Challenge 4: Security

- **Issue:** Authentication, data breaches, and fraud prevention.
- **Resolution:** Role-based access, OAuth2, encryption (TLS), and WAF.

4. Testing and Validation:

- **Unit Testing:** Jest and React Testing Library.
- **Integration Testing:** Cypress for end-to-end testing.
- **Performance Testing:** Load tests with JMeter.
- **CI/CD Tests:** Automated test pipelines with GitHub Actions.

5. Key Technologies Considered:

- **Frontend Framework:** React.js with Next.js for SSR and CSR.
- **State Management:** Redux Toolkit or Recoil.
- **Caching:** Redis and SWR.
- **API Communication:** GraphQL and REST.
- **Containerization and Orchestration:** Docker and Kubernetes.
- **Monitoring and Logs:** Prometheus and ELK Stack.

Microservices Distributed Messaging Service

1. **Clarification questions**:
    - Requirements and constraints:
        - "What are the most **critical SLA**s for this system (e.g., **latency**, **availability**)?"
        - "Are **there specific technologies or frameworks already in use**?"
        - "How does this **service interact with others** (synchronous or asynchronous)?"
2. **Problem Solving**:
    - Break down the system into smaller, manageable components and address the hardest challenges first, e.g., ensuring consistent data across distributed services or handling spikes in traffic.
3. **System Design**:
    - Explain the rationale behind each choice:
        - "I chose RabbitMQ over Kafka because we prioritize message delivery guarantees over extreme throughput."
        - "Kubernetes provides us with container orchestration and scalability."
4. **Efficiency and Scalability**:
    - Highlight specific strategies to handle scale:
        - "We'll use Redis for caching to reduce database queries."
        - "The system will autoscale using Kubernetes' Horizontal Pod Autoscaler based on CPU and memory usage."
5. **Validation**:
    - Describe how you'd test and monitor the system in production:
        - "We'll use synthetic tests to simulate user traffic and validate performance.". Including Digital Twin and JMeter
        - "Prometheus and distributed tracing will help us monitor system health/ Health Check."

B. Key pillars for consideration:

I suggest we begin with key pillars that are relevant from business point of view i.e efficiency for processing millions of booking requests per month. What do you think:

- Speed
- Scalability
- User-centric design mindset- Providing seamless experience
- QUICK Search STEPS (mention experience with BAE). Types of SEARCH algorithms. Get INTERVIEWER TO PARTICIPATE COLLABORATIVELY:

Anything else

Expanding SEARCH STEPS

1. Ranking and Relevance
2. AutoComplete and suggest
3. Aggregate SEARCH.
4. Personalisation through BROWSER History
5. Fuzzy SEARCH
6. A/B

C. HIGH LEVEL POINTERS

- **Key PILLARS in architecture (SREFOS)**
- **Key CONSIDERATIONS**
- **COMPONENTS**
- **CHALLENGES**

D. Key PILLARS in the Architecture (SSREFOST)

1. **Scalability**: Independent scaling of microservices ensures that services can handle their unique workloads without impacting others.
2. **Speed**
3. **Resilience**: Asynchronous communication and circuit breakers mitigate the risk of cascading failures.
4. **Flexibility**: A modular approach allows for iterative development, making it easier to introduce new features or update existing ones.
5. **Observability**: Centralized logging and monitoring enable debugging and performance analysis for distributed services.
6. **Security**: API Gateway and token-based authentication (OAuth2) ensure a secure entry point for external users.
7. **Testing** and **Validation**.

E. Key Considerations for Adopting Microservices away from Monolith:

Types of Architecture designs:
• SOA
• MVC
• Even Driven
• Monolith
• Microservices

Microservice*Architecture* - https://www.figma.com/board/4kkSR0KdbgOCJilUwUAVeI/Architectural-Diagrams?node-id=0-1&node-type=canvas&t=dLnC8yVc6OiJvR0u-0

The **key considerations** for adopting a Microservice were a result of bottlenecks from the Monolith architecture, which included:

Monolith

- Tight coupling
- Slow Release Cycle
- Manual Releases
- No Continuous Deployment
- Not Scalable
- Lack of log management, observation and Monitoring

Example:
The Monolithic Ruby on Rails had to perform many functionalities at the same time. Namely:

- Routing
- Data Harvesting
- Searching and Indexing
- Data persistence
- etc

No Continuous Deployment but Continuous Integration was using Travis and manual Releases using AWS Ops Works. The System was not designed to work in Containers and, hence not scalable. Many times, engineers had to intervene to scale instances in production, and there was a lack of log management and monitoring resulting in countless hours of debugging.

By leveraging tools like RabbitMQ, Amazon SQS, ElasticSearch, DynamoDB (to host jobs), Docker to host and isolate the services, Kubernetes for orchestration and scaling (detecting CPU and traffic (Cloud Watch) usage to scale horizontally adding more EC2 instances), Jenkins for deployments, IaC(Terrform to host environment configurations and migrations), NodeJS and Express Framework for API, Prometheus, ELK for Monitoring and Observability, I implemented the Microservice architectures where services communicated asynchronously, allowing them to operate independently and reliably.

Motivation for Microservice

My motivation for adopting a Microservices architecture was a result to improve the efficiency, scalability, usability, high availability, and fault tolerance of the data harvesting hub:

- Small individual components
- Easily maintainable
- Parallel development
- Rapid iteration
- Feature addition - enrichment
- Confidence in CI/CD
- Fault-tolerant
- Language-agnostic

The architecture decoupled functionalities, ensuring scalability and maintaining security and observability. RESTful API for data ingestion and transformation, message brokers (e.g., RabbitMQ) for job queues, and worker nodes for processing.

F. Key Components

Containers (**Docker**) is used for **service isolation, orchestrated and scaled** with **Kubernetes. SQS was leveraged for message queuing** between dispatchers and workers, while **ElasticSearch handled processed job indexing for real-time search.** Logging and **monitoring are centralised using Prometheus** and the **ELK** stack to ensure **visibility into system health and operations**. **Security** is reinforced with rate **limiting**, **authentication mechanisms** (e.g., OAuth), and secure communication protocols, especially for third-party integrations.

Modules:

1. **API Gateway**:
    - Acts as a single entry point for external requests.
    - Handles request routing, authentication, rate limiting, and load balancing.
    - Ensures clients interact with a unified API while hiding internal microservice complexity.
2. **Microservices Layer**:
    - Decomposed services based on specific domains (e.g., Search, Reporting, Pricing, User Management, Payments).
    - Each service owns its data and communicates with others through well-defined APIs or messaging systems.
    - Services are stateless and independently deployable.

3. **Service Communication**:
    - **Synchronous**: REST or gRPC for direct communication between services when low latency is required. Circuit Breakers to mitigate cascading of faulty messages
    - **Asynchronous**: Message brokers like Kafka or RabbitMQ for event-driven communication and decoupling.
4. **Data Layer**:
    - **Database per Service**: Each microservice has its database (SQL or NoSQL) to ensure modularity and data ownership.
    - Distributed data management strategies like eventual consistency or CQRS (Command Query Responsibility Segregation).
5. **Configuration Management**:
    - Centralized dynamic configuration through tools like Consul or etcd to manage service-specific configurations.
6. **Logging and Monitoring**:
    - Centralized logging using tools like the ELK stack (Elasticsearch, Logstash, Kibana).
    - Monitoring and observability with tools like Prometheus and Grafana to track service health and performance.
7. **Authentication and Authorization**:
    - OAuth2 or OpenID Connect for centralized identity management.
    - Role-based access control (RBAC) implemented through the API Gateway.
8. **Error Handling**:
    - Distributed error tracking (e.g., Sentry) for debugging and managing failures across services.
    - Retry mechanisms and circuit breakers to prevent cascading failures.
9. **CI/CD, Containerisation using docker to main consistency in environment dependencies and Orchestration with Scalability to support reliability of deployment and autoscaling through Health Checks by monitoring CPU usage.**
10. **Reporting Service**:
    - Decoupled from the core application and implemented as a separate service.
    - Aggregates and processes data from other microservices as needed.

G. Challenges and resolution

MSC

- ensuring **message reliability**
- **managing schema evolution**
- **maintaining consistent processing during service failures.**

Resolution :

- idempotent event handling to prevent duplicate processing, used schema registries (e.g., Confluent Schema Registry) to manage versioning
- Set up **dead-letter queues** to handle poison messages.
- Use Circuit breakers to stop the cascading of poisoned messages

Characteristics of monolithic architecture including redundant steps and unnecessary components:

LOOKING OUT FOR DEFICIENCIES FOR IMPROVEMENT!

1. **Tightly Coupled Business Logic:**
   Legacy monoliths typically have tightly coupled modules where unrelated functions are interdependent. For example, **a payment module might also handle notifications and reporting**, leading to redundant dependencies that make the system harder to scale and maintain.

2. **Centralized Databases:**
   A single database shared across all components often results in tightly coupled data models. This can lead to redundant queries and increased complexity when accessing or updating unrelated data.

3. **Overloaded APIs:**
   Monolithic systems often have large, **catch-all APIs that return excessive, irrelevant data or require multiple endpoints for simple operations, increasing redundancy and inefficiency**.

4. **Duplicated Functionality:**
   In legacy systems, similar tasks (e.g., validation, authentication) are often implemented multiple times across different parts of the application, leading to maintenance overhead and potential inconsistencies.

5. **Hardcoded Configurations:**
   Many monolithic systems rely on hardcoded configurations for workflows and connections, making it difficult to reconfigure or scale parts of the system independently.

6. **Centralized Error Handling:**
Legacy systems may have a single error-handling mechanism, making it harder to pinpoint and isolate issues within specific functionalities.

Example Unnecessary Components Found in a Migration Process:

- A *reporting/SEARCH/BOOKING module* baked into the main application instead of being decoupled as a separate service.
- Logging mechanisms tied to the monolith's main runtime, making it challenging to implement distributed logging later.
- Redundant data processing pipelines handling similar transformations across different modules.

By identifying these inefficiencies, migrating to microservices allows the system to decouple functionalities into independent, scalable services. For instance, reporting, authentication, and notification services can become separate components, each managed and deployed independently. This not only streamlines operations but also simplifies scaling and future

Monitoring and debugging are enhanced with Prometheus, Grafana, and ELK stacks to track message flow and latency. These approaches have consistently delivered resilient, high-performing systems that met real-time and high-availability requirements.

Best Practices:

- Authentication and Authorization: Implemented OAuth 2.0 for access delegation and API keys with strict scopes and expiration policies for secure access control.
- Certificate Management: Used mutual TLS (mTLS) for bidirectional authentication between the service system and third-party integrations.
- Rate Limiting and Throttling: Mitigate denial-of-service risks by enforcing rate limits on third-party API calls.
- Regular Audits: Conducted periodic security audits of third-party integrations to ensure compliance with security standards and best practices.
- Data Encryption: Sensitive data was encrypted both in transit (TLS, HTTPS) and at rest AES-256.

The outcome

The resulting architectures provide significant benefits: increased scalability with Kubernetes, better fault isolation, faster deployments through Jenkins CI/CD pipelines, and improved search capabilities with ElasticSearch.

The outcome was a robust, scalable, and secure system that ensured high availability and flexibility, enabling easier future feature integrations and updates.

Alexander Adu-Sarkodie
https://github.com/DataSolutionSoftware/Portfolio
https://github.com/kukuu?tab=repositories