SUBMITTED BY:

MUHAMMAD SAQIB

SUBMITTED TO:

SIR MUKHTIAR ZAMIN

REGISTRATION NUMBER:

FA22-BSE -005

MAJOR ARCHITECTURE PROBLEMS IN SOFTWARES:

Monolithic Architecture Scalability - Amazon (E-commerce)

Background:
Amazon initially had a **monolithic architecture**, where the entire application was built as a single codebase. Initially, this architecture was manageable, but as users and features grew, problems began to emerge.

Problem:

- **Scaling Issue**:
  - If a feature (e.g., product recommendations) needed scaling, the entire application had to be scaled.
  - This was inefficient and costly.
- **Deployment Complexity**:
  - Even for small updates, the entire application had to be redeployed.
- **Fault Isolation**:
  - If one module (e.g., the payment service) failed, the entire application would crash.

Solution:

1. **Migrated to Microservices Architecture**:
   - The application was divided into smaller, independent services.
   - Example:
     - **Cart Service**: Handles cart-related operations only.
     - **Payment Service**: Processes payments.
2. **Adopted Event-Driven Communication**:
   - Communication between services was enabled using **Amazon SQS** and **SNS** (Simple Notification Service).

3. **Containerization**:
    - Docker and Kubernetes were adopted to deploy and scale services.

Outcome:

- **Scalability**: Each service could scale independently based on its requirements.
- **Fault Tolerance**: The failure of one service did not impact the entire system.
- **Deployment Speed**: New features could be deployed quickly.

2. Real-Time Data Processing - Uber (Ride-Hailing App)

Background:
Uber’s core requirement is **real-time data processing**, where users' and drivers' data needs to be continuously synchronized. With Uber’s rapid growth, real-time data processing became a bottleneck.

Problem:

- **High Latency**:
    - It took time to locate nearby drivers.
    - Surge pricing calculations couldn’t be performed in real time.
- **Scalability**:
    - Handling asynchronous requests and real-time analytics was slow.

Solution:

1. **Lambda Architecture**:
    - Uber adopted a hybrid model of **batch processing** (for historical data) and **real-time streaming** (for real-time analytics).
2. **Apache Kafka**:
    - Kafka was used to process data streams efficiently.
3. **Dynamic Load Balancing**:
    - Load balancers were implemented to handle real-time requests.

Outcome:

- **Low Latency**: Real-time ride requests were processed more efficiently.
- **Scalable System**: The system remained responsive even as data volume increased.

3. High Latency in Content Delivery - Netflix (Streaming)

Background:
Netflix serves billions of hours of video content. Delivering content while maintaining low latency was a major issue, especially globally.

Problem:

- **High Latency**:
  - Streaming faced delays due to geographically distant servers.
- **Peak Traffic Issues**:
  - System overload occurred during new series releases.

Solution:

1. **Custom Content Delivery Network (CDN)**:
   - Netflix built its own CDN, **Open Connect**, to cache content closer to users.
2. **Dynamic Video Encoding**:
   - Video quality was dynamically adjusted based on the user's internet bandwidth.

Outcome:

- **Seamless Streaming**: Latency was reduced, and buffering issues were almost eliminated.
- **Cost Savings**: Open Connect reduced dependency on third-party CDNs.

4. Security Vulnerabilities in API - Twitter (Social Media)

Background:
Twitter APIs are open to all kinds of developers, which led to an increase in bot activity and spam attacks.

Problem:

- **Unauthorized Access**:
  - Lack of proper authentication was a major security flaw.
- **Abuse of API**:
  - Bots misused APIs to create fake accounts and spam content.

Solution:

1. **OAuth 2.0**:
   - Implemented secure authentication and token-based authorization.
2. **Rate Limiting**:
   - A limit was set on the number of requests per user per second to control abuse.
3. **API Gateway**:
   - A centralized API gateway was deployed to filter and monitor incoming requests.

Outcome:

- **Improved Security**: Unauthorized access and bot activity were significantly reduced.
- **System Stability**: System crashes caused by API abuse were minimized.

5. Fault Tolerance in Distributed Systems - Spotify (Music Streaming)

Background:
Spotify relies heavily on distributed services, and the failure of one service could potentially impact the entire application.

Problem:

- **Single Point of Failure**:
  - If a server or service went down, the entire system would be affected.
- **Data Loss Risk**:
  - There was no fallback or redundancy setup.

Solution:

1. **Circuit Breakers**:
   - Used libraries like Spring Cloud Resilience4j and Netflix Hystrix to prevent cascading failures.
2. **Geo-Redundancy**:
   - Deployed servers across multiple regions to ensure availability.
3. **Chaos Engineering**:
   - Simulated intentional failures to identify weak points.

Outcome:

- **High Availability**: Service downtimes were significantly reduced.
- **Resilience**: Chaos engineering prepared Spotify for unexpected failures.