SOFTWARE DESIGN AND ARCHITECTURE

USMAN ALI AWAN

FA21-BSE-159

DATE: 31-12-2024

ASSIGNMENT 2

TASK:

As you know that software version is updated whenever we face any architectural changes. These kinds of architectural changes have major impacts on the system. In the first part, You need to find 5 major architectural problems which were faced in software system and then either the module or the whole system was revamped to solved the issue or a set of issues. In the second part you need to replicate one problem and then solve it with the help of coding example.

You need to submit a link pointing to a folder on your GitHub repository that contains your PDF report and the coding example in running form.

SOLUTION:

PART 1 : Architectural Problems in Software Systems

Following are the 5 major problems:

1. Scalability Issues

Problem: As the user base grows, a software system may struggle to handle the increased load, leading to performance degradation, high latency, and system crashes. **Solution**: The system may undergo a complete architectural overhaul, including adopting horizontal scaling, load balancing, and microservices architectures. For instance, a monolithic application may be split into microservices to scale independently. **Example**: The migration of Instagram from a monolithic system to a microservices-based architecture to handle its massive growth.

2. Tight Coupling Between Components

Problem: When components or services are tightly coupled, changes to one part of the system can break other parts, making maintenance challenging and slowing down development. **Solution**: A refactor to introduce loose coupling via dependency injection, microservices, or service-oriented architecture (SOA) to ensure that individual services can evolve independently

without affecting the entire system. **Example**: The transition of Amazon's architecture from tightly coupled systems to microservices to facilitate faster development cycles.

3. Single Point of Failure (SPOF)

Problem: When a critical component or service becomes a bottleneck or fails, the entire system crashes or experiences downtime, resulting in service outages. **Solution**: Redundant systems, failover mechanisms, and distributed architectures such as clustering, replication, or backup systems are implemented to eliminate SPOFs. **Example**: Netflix transitioned to a distributed, fault-tolerant cloud infrastructure using AWS and Chaos Monkey to eliminate SPOF.

4. Lack of Maintainability and Code Duplication

Problem: As software systems grow, the code base can become difficult to maintain due to duplication and lack of modularity, which also leads to inconsistent behavior and harder debugging. **Solution**: Refactoring to improve code modularity, using design patterns, and applying principles like DRY (Don't Repeat Yourself) can help. Automated testing and continuous integration also improve maintainability. **Example**: Refactoring of legacy systems at Microsoft to improve code modularity and maintainability through service decomposition.

5. Poor Performance and Latency

Problem: Systems that rely on synchronous processes or have inefficient data access patterns (e.g., frequent database queries) can face performance bottlenecks, leading to poor user experience and high latency. **Solution**: Implementing asynchronous communication, optimizing database queries, and introducing caching mechanisms can resolve these issues. Additionally, moving to event-driven architectures can help decouple processes and improve responsiveness. **Example**: LinkedIn's shift from a monolithic system to a microservices architecture and its use of Kafka to improve performance and scalability.

PART 2: Replicating One Problem and Solving It

Example Code:

Before applying DI, the classes are tightly coupled.

```
class Database:
    def connect(self):
        return "Connected to Database"


class UserService:
    def __init__(self):
        self.db = Database()  # Tightly coupled to Database
```

```
    def get_user(self, user_id):
        if self.db.connect():
            return f"Fetching user {user_id}"


# Usage
user_service = UserService()
print(user_service.get_user(123))
```

Refactor using Dependency Injection:

In this refactored version, UserService no longer creates the Database instance directly. Instead, it receives a Database object via its constructor.

```
class Database:
    def connect(self):
        return "Connected to Database"


class UserService:
    def __init__(self, db: Database):  # Dependency Injection
        self.db = db

    def get_user(self, user_id):
        if self.db.connect():
            return f"Fetching user {user_id}"


# Usage
db = Database()  # Database is created outside
user_service = UserService(db)  # Database is injected
print(user_service.get_user(123))
```