COMSATS University
Islamabad
Abbottabad Campus

SUBJECT SDA:

SUBMITTED BY

NAME TAHIR ALI

REG NO: FA22-BSE-050

SECTION: 5A

SUBMITTED TO:

SIR MUKHTIAR ZAMIN

Software Architecture Report:

Major Architectural Problems in Software Systems:

Software architecture plays a critical role in the performance, scalability, and maintainability of a system. Over time, systems evolve, and changes in architecture become necessary. Below are **five major architectural problems** that software systems face, along with solutions based on industry-standard principles like **Clean Architecture** and **Micro services**:

1. Scalability Problems in Monolithic Systems:

Problem: Monolithic applications are difficult to scale because all components are tightly coupled. When demand increases, scaling becomes inefficient.

Solution: The system was split into micro services. Each service is independently scalable and handles a specific business functionality, allowing the system to scale efficiently based on load.

2. Single Point of Failure in Monolithic Systems:

- **Problem:** In monolithic systems, a failure in one part of the system can bring down the entire application.
- **Solution:** Micro services architecture was adopted, where each service is isolated. This means that if one service fails, others can continue running, minimizing the impact of failure.

3. Difficult Maintenance in Legacy Systems:

- **Problem:** Legacy systems are difficult to maintain due to their large, complex codebases. Changes in one part of the system often require extensive testing and affect other parts.
- **Solution:** The system was refactored into smaller, more modular components. This modular approach made the code easier to maintain, test, and extend without affecting the entire system.

4. Database Bottlenecks in Large Systems:

- **Problem:** A centralized database can create performance bottlenecks when large volumes of data are being processed.
- **Solution:** A distributed database architecture was implemented, where the data was partitioned and stored across multiple nodes, improving data access and system performance.

5. **Slow Development Cycles Due to Tight Coupling:**
    - **Problem:** In tightly coupled systems, changes in one module require changes to others, slowing down the development process.
    - **Solution:** The system was redesigned to use a **Service-Oriented Architecture (SOA)**, where modules are loosely coupled, allowing development teams to work independently and improving the speed of deployment.

Replicating and Solving an Architectural Problem:

Problem: Scalability Issue in Monolithic Systems:

In a monolithic application, all components are interdependent, leading to scalability issues when the application grows in size and complexity. Let's replicate this problem and solve it using a simple Java example.

Problem Replication:

We assume we have a monolithic application where a User Service handles all user-related functionalities and a Product Service handles product operations. Both services are tightly coupled and share a single database connection, causing performance issues when scaling.

Solution: Refactoring the Monolithic System to Micro services:

To solve this issue, we break the system into smaller, independent micro services. Each service will handle a specific business function, and they will communicate with each other through API calls.

Solving Tight Coupling with Clean Architecture Principles:

Problem: Tight coupling occurs when different components of the system depend on each other too heavily, making the system harder to extend and modify.

Solution: Refactor with Clean Architecture:

1. **Interfaces:** Define interfaces (e.g., IUser DAO, IProduct DAO) to decouple the data access logic from the business logic.

2. **Dependency Injection:** Use dependency injection to inject DAO instances into services, instead of directly creating them inside the service class. This allows for greater flexibility and testability.

3. **Separation of Concerns:**

   Keep business logic and data access logic separate. This follows the principle of **Separation of Concerns**, making the system more maintainable.

Conclusion:

By applying **Clean Architecture** and refactoring the code to remove **Tight Coupling**, the system is made more flexible, maintainable, and scalable. These changes allow future extensions to be implemented without breaking existing code, making it easier to add new features and maintain the system over time. The introduction of **Microservices** also provides better scalability, fault tolerance, and development flexibility.