SDA LABS

Muhammad Abubakar (FA22-BSE-070)

FIVE MAJOR ARCHITECTURAL PROBLEMS WITH THEIR SOLUTIONS:-

1. Inadequate Requirements Gathering (Architectural Problem)

Problem:

The system's architecture lacked flexibility, making it difficult to adapt to changing or evolving requirements. The design may have been monolithic or tightly coupled, which made adding new features or modifying existing ones complex and error-prone.

Solution:

- **Microservices Architecture:** The system was restructured into smaller, independent services that could be developed, deployed, and scaled independently, making it easier to handle evolving requirements.
- **Modular Design:** Each component or service was designed to handle a specific function, ensuring that changes could be made in one area without disrupting the whole system.
- **Domain-Driven Design (DDD):** The system was aligned with business goals by breaking it down into logical domains, ensuring that the architecture was flexible to accommodate future changes in business requirements.

2. Poor Communication and Collaboration (Architectural Problem)

Problem:

The architecture caused silos within the development teams, making it hard for teams to collaborate and work on different parts of the system without impacting each other. Tight coupling between components prevented independent development.

Solution:

- **Microservices Architecture:** The system was divided into smaller, independent services with well-defined boundaries. Each team could focus on a specific service, allowing them to work autonomously without affecting other teams.
- **Clear API Boundaries:** APIs were established between services, ensuring that each service could interact with others in a well-defined, decoupled way. This streamlined communication and collaboration across teams.

3. Scope Creep (Architectural Problem)

Problem:

The architecture wasn't designed to scale or handle additional features effectively, leading to scope creep. As new requirements emerged, the rigid architecture couldn't accommodate them easily, forcing frequent rework and continuous changes to the core structure.

Solution:

- **Modularization:** The architecture was broken down into smaller, more manageable modules, which could be enhanced or expanded independently. This made it easier to accommodate new features without disrupting the entire system.
- **Microservices Architecture:** The use of microservices allowed for incremental changes, where new functionality could be added by introducing new services without impacting the existing ones.

4. Technical Debt (Architectural Problem)

Problem:

The system had accumulated technical debt, where shortcuts were taken during initial development to meet deadlines, resulting in a messy, complex, and unmanageable codebase. Over time, this made it difficult to maintain and scale the system.

Solution:

- **Clean Architecture:** The system was refactored using clean architecture principles, ensuring separation of concerns, clear boundaries between layers (data, business logic, user interface), and a modular structure that promotes maintainability and scalability.
- **Layered Architecture:** The system was structured in layers, with each layer responsible for a specific concern, ensuring that changes in one layer wouldn't affect others.
- **Dependency Injection:** Aimed at reducing tightly coupled components, dependency injection allowed for more flexible and testable code, making it easier to refactor and manage the system in the long run.
- **Continuous Refactoring:** A process of regularly revisiting and improving the codebase was introduced to reduce technical debt and maintain code quality.

5. Inadequate Testing (Architectural Problem)

Problem:

The system was not designed to support effective testing. Tight coupling between components made it difficult to isolate and test individual parts of the system. As a result, bugs were harder to identify, and regression testing was complicated.

Solution:

- **Clean Architecture:** The architecture was redesigned to support testing by ensuring that business logic and external dependencies were clearly separated. This made it easier to test individual components in isolation.
- **Hexagonal Architecture (Ports and Adapters Pattern):** This pattern allowed the core logic of the system to be decoupled from external dependencies (like databases, APIs, or frameworks), facilitating easier testing of the core application logic.
- **Test-Driven Development (TDD):** TDD was adopted as a development practice, ensuring that tests were written before the code itself, which led to better test coverage and fewer defects.
- **Continuous Integration (CI):** CI practices were introduced to automate the testing process and integrate code regularly into a shared repository, allowing early detection of issues and ensuring that the system remains stable as new changes are introduced.