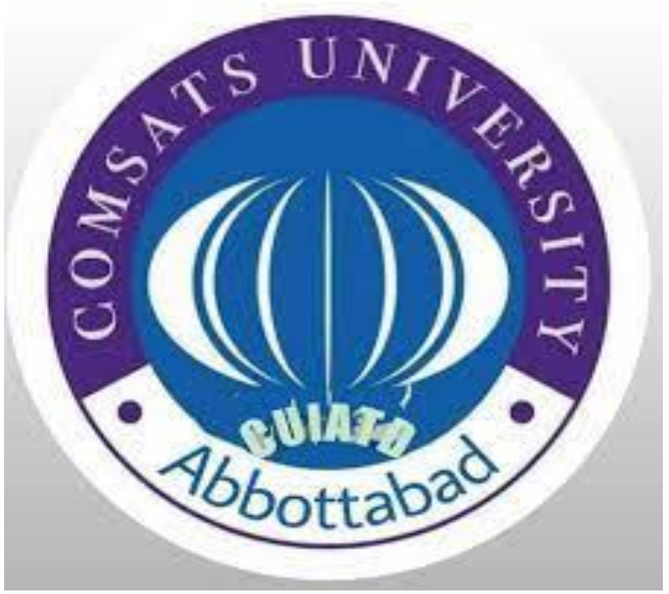

Submitted by

Ahmad Ayyar khan

Submitted to :

Sir mukhtair Zamin

Registration No

SP23-BSE-021

Problem:

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

Solution:

Five Major Architectural Problems in Software Systems

---

1. Monolithic Architecture

Problem:

A monolithic architecture is one in which the entire application is developed and deployed as a single, indivisible unit. All the business logic, UI, and data access layers are tightly packed into one large codebase. This setup works well for small-scale applications but becomes a serious issue as the system grows. The primary concerns include:

- **Scalability:** Scaling the application means scaling everything together, even if only one part needs more resources.
- **Maintainability:** Making changes to one module can unintentionally affect others. Debugging becomes more complex and risk-prone.
- **Flexibility:** Introducing new technologies or frameworks is difficult because the whole system depends on the current stack.

Solution: Microservices Architecture

To address these limitations, the system is often **revamped into Microservices Architecture**. Microservices divide the monolith into smaller, self-contained services that can be developed, tested, deployed, and scaled independently. Each microservice handles a specific business function and communicates with others via APIs.

Example:

A banking application with modules like:

- Account management
- Transactions
- Customer support

Initially built as a monolith, it became hard to manage. By converting it into microservices:

- The **transaction service** could be independently scaled for high traffic.
- **Bug fixes** in the customer support module didn't require redeployment of the entire application.
- Teams could work in parallel on different modules using different technology stacks if needed.

---

2. Scalability Issues

Problem:

As the number of users grows or application features expand, the system may **fail to meet performance expectations**. This typically happens in systems that were designed without scalability in mind. Symptoms include:

- Increased **response time**
- System **crashes** during peak loads
- **Resource exhaustion** (CPU, memory, bandwidth)

Solution: Distributed System Architecture

To solve scalability issues, systems are restructured into **distributed architectures**. Key solutions include:

- **Horizontal Scaling**: Adding more machines instead of upgrading a single one.
- **Load Balancing**: Distributing requests evenly among multiple servers.
- **Cloud Integration**: Leveraging cloud platforms like AWS or Azure for elastic resource provisioning.

Example:

An e-commerce platform originally serving a few hundred users now serves millions. To manage this:

- A **load balancer** was added to distribute traffic.
- **Product catalog**, **order management**, and **payment processing** were split into separate services.
- Cloud services allowed automatic scaling during festive sales.

---

3. Tight Coupling Between Components

Problem:

In tightly coupled systems, modules are highly dependent on one another. This creates problems like:

- **Low flexibility** in updating or replacing modules.
- A small change in one component causes **cascading changes** in others.
- Difficulty in **testing** and **continuous deployment** due to interdependencies.

Solution: Event-Driven Architecture (EDA) and **Message Queues**

An **event-driven architecture** enables decoupled communication where components react to events asynchronously, rather than calling each other directly. This is usually achieved via **message brokers** such as:

- **Apache Kafka**
- **RabbitMQ**

Example:

An ERP system with modules for:

- Finance
- Human Resources
- Inventory

Initially, these were directly integrated, causing ripple effects from any changes. After adopting **event-driven architecture**:

- When a new employee was added in HR, an event was published.
- The Finance module subscribed to the event and updated payroll independently.
- **No direct coupling** existed between the services.

---

4. Poor Performance Due to Database Bottlenecks

Problem:

When all modules or services rely on a single, centralized database, performance suffers:

- High **latency** during peak operations

- **Deadlocks** or **race conditions**
- Limited ability to **scale reads/writes**

Solution: Caching, Sharding, and NoSQL Databases

To fix database performance issues:

- **Caching** (e.g., Redis, Memcached) stores frequently accessed data in memory.
- **Sharding** splits the database into smaller pieces distributed across servers.
- **NoSQL databases** (e.g., MongoDB, Cassandra) are introduced for specific use cases like analytics or document storage.

Example:

A content management system (CMS) using a single SQL database slowed down during peak hours. To optimize:

- Frequently accessed pages were cached in **Redis**.
- User data was sharded by geographic region.
- Blog content was migrated to **MongoDB** for flexible querying.

---

5. Security Vulnerabilities in Legacy Systems

Problem:

Legacy systems often use outdated security models and software, making them vulnerable to:

- **SQL injection**
- **Cross-site scripting (XSS)**
- **Authentication flaws**
- **Data breaches**

These issues occur because of:

- Lack of **data encryption**
- Weak or **hardcoded credentials**
- Absence of **input validation**

Solution: Modern Security Practices

To enhance security:

- Implement **OAuth2/OpenID** for secure authentication.
- Use **encryption standards** like AES for data at rest and TLS for data in transit.
- Enforce **role-based access control (RBAC)**.
- Add **Two-Factor Authentication (2FA)**.

Example:

A hospital management system suffered a data breach due to weak password storage and lack of encryption. The system was updated to:

- Use **OAuth2** for authentication.
- Encrypt patient records using **AES-256**.
- Enforce **2FA** for all staff logins.
- Implement **security audits** and logging mechanisms.