<u>NAME: ALI BASHIR</u>

<u>SUBMITTED TO: SIR MUKHTIAR ZAMIN</u>

<u>REG NO: FA22-BSE-007</u>

<u>LAB ASSIGNMENT 2</u>

Major Architectural Problems in Software Systems

As in this assignment we will talk about some software that was initially launch with architecture failures which was later get solved but they had architecture problems these are as follows:

1. **Denver International Airport Baggage System (1995)**
   Problem:
   The ambitious plan to automate the entire baggage system ended in failure due to unmanageable complexity and lack of proper testing. The project caused a 16-month delay and cost overruns of $560 million.

Architectural Issues in Version 1.0:

- **Centralized Control:** A single control system made the entire system vulnerable to breakdowns.
- **Poor Modularity:** The terminals were interdependent, meaning a problem in one affected the whole system.
- **Weak Error Handling:** There were no backup processes to manage baggage jams or route errors.

Resolution in Version 2.0:

- **Decentralized Design:** Each terminal was equipped with its own partially automated system to work independently.
- **Simplified Processes:** Automation was reduced to allow manual intervention when needed.
- **Thorough Testing:** The updated system was tested in smaller parts before full deployment.

2. **Healthcare.gov Launch Issues (2013)**
   Problem:
   The launch of the Affordable Care Act's website was plagued by crashes, incomplete sign-ups, and slow performance, especially under high traffic.

Architectural Issues in Version 1.0:

- **Fragmented Development:** Different contractors developed isolated components without proper integration.
- **Scalability Problems:** The servers couldn't handle the sudden influx of users.
- **No Real-World Testing:** The system wasn't tested for the scale it faced at launch.

Resolution in Version 2.0:

- **Scalable Architecture:** Moved to a cloud-based system with elastic scaling to handle surges.

- **Centralized Coordination:** An expert team managed integration and streamlined the overall system.
- **Load Testing:** Conducted tests simulating real-world conditions to identify and fix bottlenecks.

Outcome:
Within six weeks, the updated system could manage over 200,000 users at once without crashing.

3. <u>**Twitter's Early Scaling Challenges (2008-2010)**</u>
   Problem:
   As Twitter's user base grew rapidly, its initial system couldn't keep up, resulting in frequent outages symbolized by the infamous "Fail Whale."

Architectural Issues in Version 1.0:

- **Monolithic Design:** All features were tightly connected, making scaling specific parts difficult.
- **Single Database Bottleneck:** Heavy reliance on one database caused slowdowns.
- **No Traffic Redundancy:** The system lacked proper load-balancing mechanisms.

Resolution in Version 2.0:

- **Microservices:** The monolith was split into smaller, independent services.
- **Distributed Databases:** Adopted scalable NoSQL databases for handling large data volumes.
- **Caching and Redundancy:** Added caching layers and load balancers to distribute traffic.

Outcome:
The revamped architecture handled major events like World Cups without downtime.

4. <u>**London Ambulance Service Dispatch Failure (1992)**</u>
   Problem:
   A new automated dispatch system led to severe delays in sending ambulances, putting lives at risk.

Architectural Issues in Version 1.0:

- **Over-Centralization:** The entire system depended on one central server.
- **Real-Time Limitations:** It couldn't process real-time data like traffic and locations efficiently.
- **Inadequate Testing:** The system wasn't tested under real-world conditions.

Resolution in Version 2.0:

- **Simplified Design:** Focused on automating essential parts and kept manual control for others.

- **Incremental Deployment:** Rolled out in phases to ensure stability at each stage.
- **Improved Real-Time Processing:** Enhanced algorithms to handle live traffic and location data.

Outcome:
The updated system improved ambulance response times and reliability.

5. **Boeing 737 MAX MCAS Failures (2019)**
   Problem:
   The MCAS system relied on a single sensor, and sensor failure caused fatal crashes.

Architectural Issues in Version 1.0:

- **Single Point of Failure:** Only one sensor fed critical data to the MCAS.
- **Limited Pilot Awareness:** Pilots weren't fully informed about MCAS operations.
- **Insufficient Testing:** Edge cases involving sensor failures weren't addressed.

Resolution in Version 2.0:

- **Redundancy Added:** Multiple sensors were used to ensure accuracy.
- **Pilot Overrides:** Pilots could disable MCAS in emergencies.
- **Comprehensive Testing:** Rigorous simulations and real-world testing ensured safety.

Outcome:
The system met safety standards and allowed the 737 MAX to return to service in 2021.