# SDA LABS

**Muhammad Abubakar (FA22-BSE-070)**

## Architectural Problem: Scope Creep

**Problem Definition:**

**Scope creep** refers to the uncontrolled changes or continuous growth in a project's scope, often without adjustments to budget, timeline, or resources. This phenomenon can occur when new features or requirements are added to the system after the project has started, which can lead to project delays, budget overruns, and software that becomes too complex or difficult to maintain.

In the context of software development, scope creep usually results from unclear requirements, changing customer needs, or a lack of formal process to handle feature additions during development. When these changes aren't carefully managed, the architecture can become more complex than originally designed, leading to issues like **over engineering**, **increased technical debt**, and **loss of focus on the core features** of the system.

## How Scope Creep Occurred:

**1. Initial Monolithic Architecture Design:**

Initially, the software was likely developed using a **monolithic architecture**. In a monolithic design, all features of the software are bundled together into one tightly integrated unit. While this can work for simple applications, it can become problematic as the system grows in size and complexity.

- **Symptoms of Scope Creep**:
  - **Unanticipated New Features**: As the project progressed, stakeholders or customers requested additional features that were not part of the original project scope. The monolithic architecture made it difficult to incorporate these features seamlessly.
  - **Tight Coupling**: The tightly coupled nature of monolithic systems made it hard to modify or extend one part of the system without affecting others, which led to unintended consequences and delays.
  - **Increased Complexity**: Each additional feature added to the system increased the overall complexity of the monolithic codebase, which made it harder to maintain and modify. As new requirements kept emerging, the system became bloated.

**2. Lack of Defined Boundaries Between Features:**

In a monolithic system, the boundaries between features are often unclear, leading to interdependencies that make it hard to introduce new features without touching the core system. As requirements changed, it became difficult to properly assess the impact of adding new features.

- **Symptoms of Scope Creep**:
    - The system's boundaries weren't clearly defined, making it easy for teams to add new features without considering their impact on other parts of the system. Each new feature had to be integrated into the central system, which created unforeseen challenges as the software grew.

**3. Delayed Project Deadlines:**

As new features were added continuously, the project started to slip from its original timeline. The unanticipated complexity of the monolithic architecture caused delays, and the new features increased the time required for integration and testing.

- **Symptoms of Scope Creep**:
    - A constantly shifting deadline with unclear scope, leading to team frustration and an inability to deliver a product on time.

## How Scope Creep Was Resolved:

**1. Transition to a Microservices Architecture:**

The primary architectural solution to combat scope creep was the transition from a **monolithic architecture** to a **microservices architecture**. Here's how this shift helped:

- **Decoupling Services**: Microservices are small, independent services that are focused on a single functionality. Each service can be developed, deployed, and scaled independently. By breaking down the monolith into individual services, it became easier to add new features as separate services, reducing the impact on the core system.
    - **Benefits**:
        - Each service can be developed and deployed independently, reducing the risk of unwanted side effects.
        - New features can be introduced without affecting the core system, reducing integration complexities.
- **Clear Boundaries Between Features**: Microservices define clear API boundaries between each service. This made it possible to assess the impact of new features more effectively, ensuring that they could be added in a controlled and structured way.
    - **Benefits**:
        - New features can be added by simply introducing new services that don't interfere with existing ones.
        - Each service can be updated or replaced independently as requirements evolve, which reduces the risk of scope creep.

**2. Modularization and Feature Flags:**

To deal with ongoing scope creep, the development team introduced a **modular approach** to the design, allowing for smaller, more manageable components that could be iteratively developed and released.

- **Modularization**: By decomposing the system into clear, functional modules, each module could be developed separately, allowing the team to focus on core features while ensuring flexibility for future additions.
    - **Benefits**:
        - Each module could be worked on independently, enabling teams to prioritize key features without overwhelming the system.
        - This allowed teams to better handle changing requirements while keeping the overall project under control.
- **Feature Flags**: Feature flags (also known as toggles) were introduced to control the rollout of new features. This allowed the team to release code into production without fully activating the new feature until it was ready.
    - **Benefits**:
        - New features could be tested in production without affecting all users, reducing the risk of introducing bugs or performance issues.
        - It became easier to manage scope creep by activating only the features that were truly ready and needed, while deferring others.

**3. Clearer Requirement Management & Change Control:**

A formalized process for handling changing requirements was introduced, along with regular reviews to ensure that the scope of the project was clearly defined and agreed upon by all stakeholders.

- **Change Control Process**: A change control board (CCB) was set up to review any requests for new features or modifications. Each request was carefully assessed for its impact on the project, including cost, timeline, and complexity.
    - **Benefits**:
        - This ensured that no new features were added without proper evaluation.
        - Only essential features were added, reducing unnecessary complexity and keeping the system aligned with the original vision.
- **Agile Methodology**: An Agile framework (e.g., Scrum or Kanban) was adopted to help manage scope creep in a more controlled and iterative way. This allowed for regular reviews, sprint planning, and better communication between stakeholders and developers.
    - **Benefits**:
        - Iterative development meant features could be released in small chunks, allowing for more flexibility while keeping the project on track.
        - Frequent feedback from stakeholders helped ensure the development was focused on the most important features.

**4. Proper Resource and Time Allocation:**

As scope creep continued to stretch deadlines, it became evident that resources and time needed to be allocated more effectively. This was achieved by **setting up clear timelines** and assigning specific teams or resources to work on defined features or services.

- **Dedicated Teams for Specific Services**: Each microservice was assigned to a dedicated team that was responsible for its development and maintenance. This created better focus and allowed the team to avoid scope creep within individual services.
    - **Benefits**:
        - With dedicated resources, teams could focus on individual services, which made it easier to handle feature additions without introducing delays or confusion.

## Summary of Solution to Scope Creep:

| Aspect | Solution |
|---|---|
| **Architecture** | **Microservices Architecture** – breaking down monoliths into independent, scalable services with clear boundaries and interfaces. |
| **Modularity** | **Modularization** – decomposing the system into manageable functional components. |
| **Feature Rollout** | **Feature Flags** – controlling the release of features and testing in production. |
| **Requirement and Change Management** | **Change Control Process** – formalizing the process of handling new requirements and modifications. |
| **Development Process** | **Agile Methodology** – iterative development with regular reviews and prioritization. |
| **Resource Allocation** | **Dedicated Teams for Services** – ensuring focused development on specific microservices. |

By adopting these architectural practices, scope creep was effectively controlled. The system became more flexible, easier to maintain, and capable of evolving with changing requirements without overwhelming the original architecture.