CI/CD Pipeline Implementation with Student Management System

(Design Document)

Reza Asar (23501688)

Eastern Mediterranean University

CMSE 520: Software Evolution and Maintenance

Prof. Dr. Alexander Chefranov

Nov 30, 2024

Contents

Introduction

This design document presents the technical architecture and detailed design decisions for implementing a CI/CD pipeline demonstration project. The project uses a simple Student Management System as a practical example to showcase modern CI/CD practices and tools.

This document is structured in two main parts:

1. Application Design: Detailed technical design of the Student Management System, including architectural decisions, data structures, and user interfaces.
2. CI/CD Pipeline Design: Mapping of our CI/CD requirements to specific implementation decisions, showing how each requirement will be fulfilled through our chosen tools and processes.

While the Student Management System is intentionally kept simple with basic CRUD operations, the emphasis is on designing a robust and scalable CI/CD pipeline that demonstrates industry-standard practices. Each section will explicitly connect design decisions back to the requirements specified in the SRS document.

Application Design

Architectural Design Overview

The Student Management System implements Clean Architecture principles (Martin, 2017), ensuring separation of concerns and domain-centric design. The solution structure enforces dependency rules flowing inward, with the domain layer at the core. This architecture is complemented by Repository and Unit of Work patterns (Evans, 2003) for data access management.

Key architectural layers (from innermost to outermost):

- Domain Layer: Core business entities and logic
- Application Layer: Application services and interfaces
- Infrastructure Layer: Data access implementation, external services
- Presentation Layer: API endpoints and Angular frontend

Application of Clean Architecture in Student Management Context

In our Student Management System, Clean Architecture principles manifest as follows: The Domain Layer contains the *Student* entity with essential properties. The Application Layer defines IStudentService for core operations like enrollment and retrieval, while StudentService implements these using domain entities. The Infrastructure Layer implements StudentRepository and UnitOfWork for data persistence through Entity Framework Core. Finally, the Presentation Layer exposes these capabilities through WebAPI endpoints consumed by the Angular frontend, maintaining clear separation of concerns throughout.

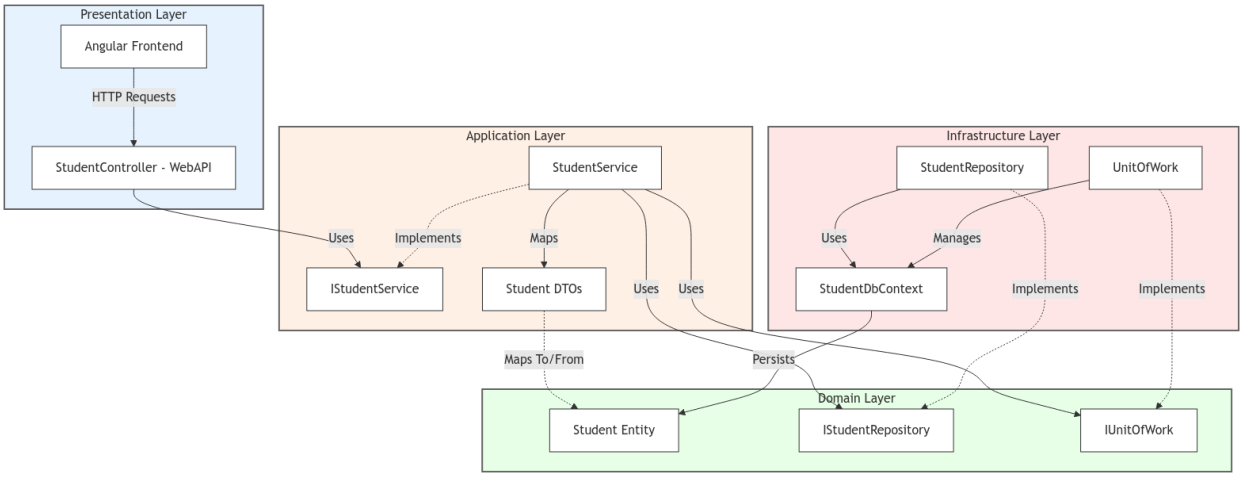

Figure 1- Clean Architecture - Student Management System

Class Structure

The class diagram illustrates the core entities and interfaces of our Student Management System. The design emphasizes simplicity while maintaining clean architecture principles. The Student

class serves as our primary domain entity, containing essential student information. The interfaces IStudentRepository and IUnitOfWork define the contracts for data operations and transaction management, respectively, adhering to the repository pattern and maintaining separation of concerns.

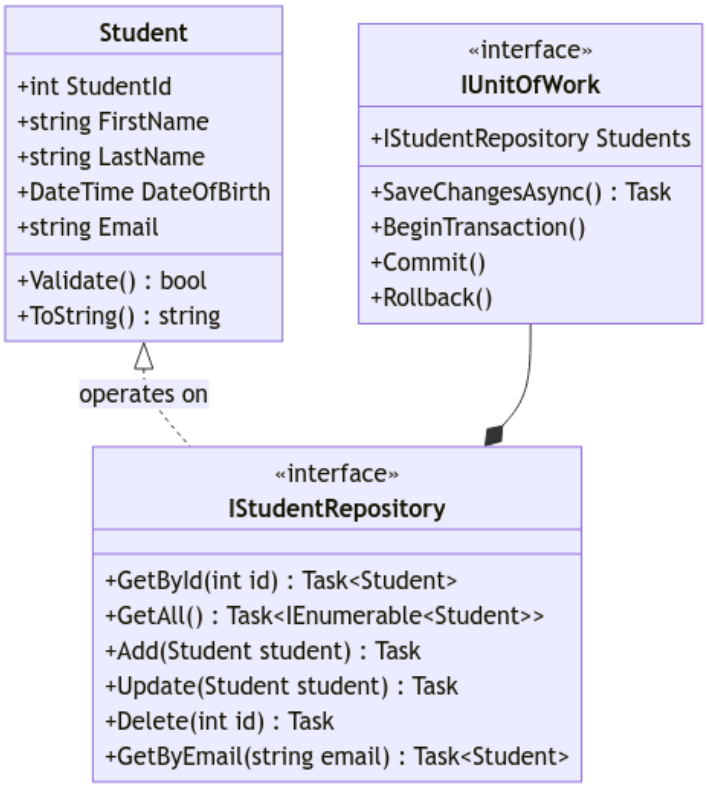

Figure 2- Class Diagram - Student Management System

Database Design

The database schema follows a straightforward design that aligns with our domain model. While maintaining simplicity for demonstration purposes, the design ensures data integrity and supports all required CRUD operations. The Student table serves as our primary entity store, with fields directly mapping to our domain model properties.

Student			
int	StudentId	PK	PRIMARY KEY IDENTITY(1,1)
nvarchar(50)	FirstName		NOT NULL
nvarchar(50)	LastName		NOT NULL
date	DateOfBirth		NOT NULL
nvarchar(100)	Email		NOT NULL UNIQUE

Figure 3- Database Schema with Data Types and Constraints - Student Management System

Interface Design

The user interface design focuses on providing a clean, intuitive experience for managing student records. The design follows a standard CRUD pattern, with separate views for listing, creating, updating, and deleting student records. The interface emphasizes simplicity and usability while maintaining a professional appearance.

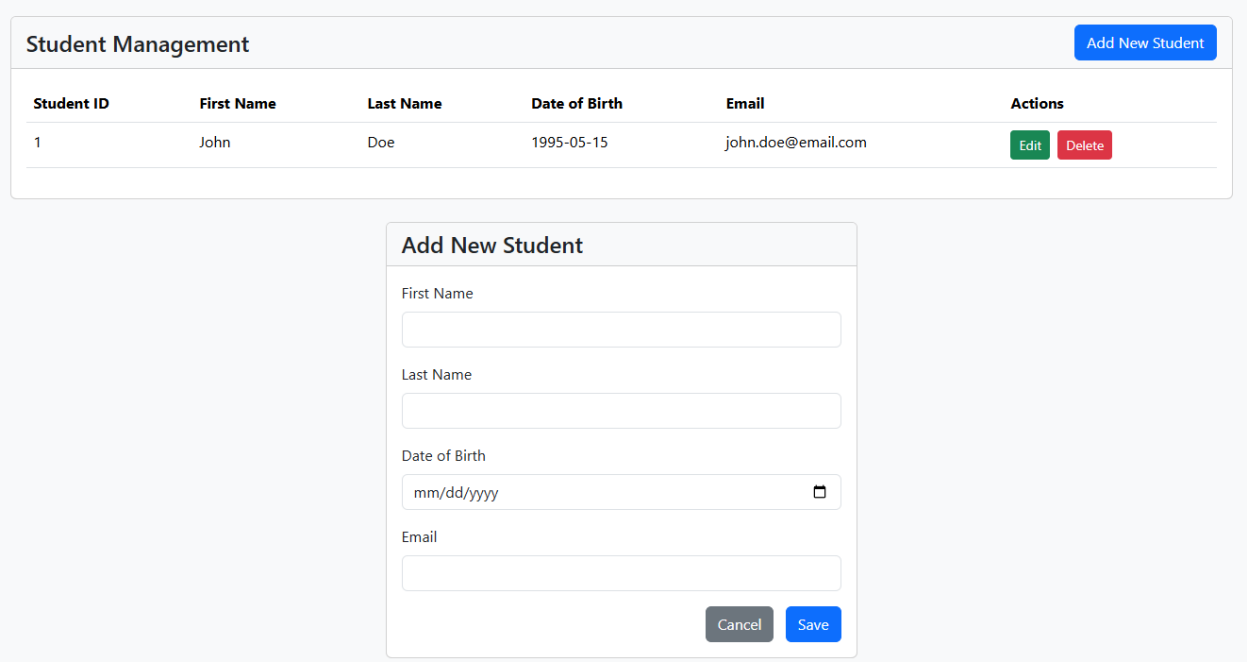

Figure 4- Student Management System Interface Views

CI/CD Pipeline Design

Requirements to Implementation Mapping

This section demonstrates how each CI/CD requirement maps to specific implementation decisions and tools.

Table 1- Mapping of CI/CD Requirements to Implementation Decisions and Tools

Category	SRS Requirement	Implementation	Tool Choice
Source Control Management	The system must automatically trigger pipeline processes when code is committed	GitHub repository configured with GitHub Actions triggers on push events	GitHub for version control and automated triggers
Build and Test	The system must automatically build the application and execute tests	Multi-stage pipeline that builds .NET backend and Angular frontend, followed by XUnit tests	GitHub Actions for orchestration, XUnit and Moq for testing
Database Integration	The system must execute database migrations during deployment	EF Core migrations included in deployment process, executed before application deployment	EF Core CLI tools integrated into pipeline
Containerization	The system must create Docker containers for both frontend and backend	Multi-stage Dockerfiles for optimized container builds	Docker for containerization, Docker Compose for local orchestration
Deployment	The system must deploy the containerized application to local Kubernetes	Kubernetes manifests for deployments, services, and configurations	Local Kubernetes cluster (via Docker Desktop)

Additional Requirements	Requirements related to coding standards, documentation, and long-term maintainability	Several requirements, particularly those related to coding standards, documentation, and long-term maintainability, are addressed through architectural decisions and implementation guidelines rather than specific tool configurations

Pipeline Flow Design

This section illustrates the complete flow of our CI/CD pipeline, from initial code commit to final deployment. The pipeline is designed to ensure code quality, maintain system stability, and automate the entire deployment process. Each stage represents a crucial step in the delivery process, with built-in quality gates and rollback mechanisms to handle failures gracefully.

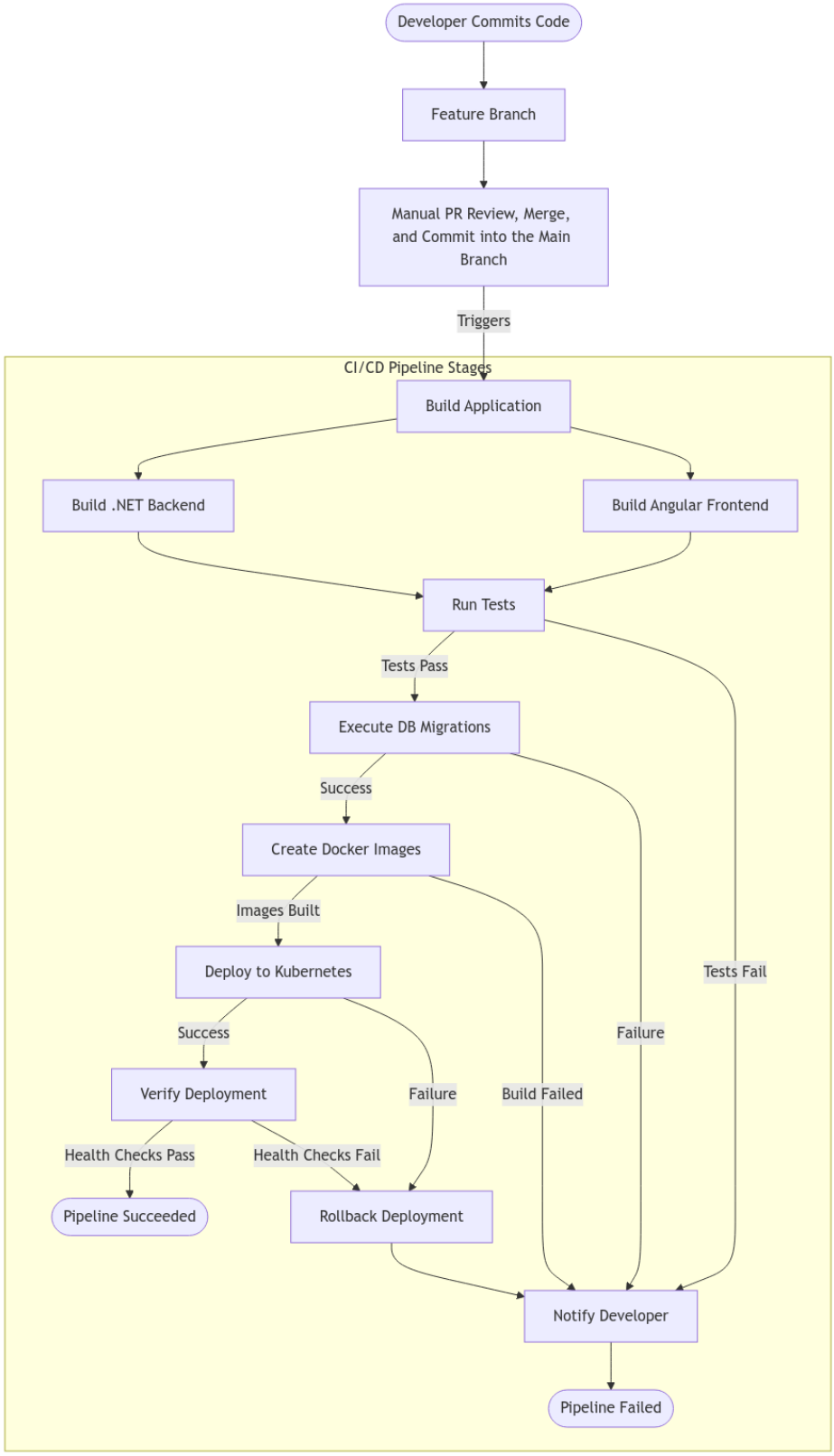

Figure 5- CI/CD Pipeline Flow: From Code Commit to Deployment with Quality Gates and Rollback Mechanisms

Continuous Integration Strategy (CI)

This section outlines our approach to implementing Continuous Integration, ensuring code changes are reliably integrated and validated. The CI process begins when developers commit code and encompasses:

- **Code Integration:** The integration process begins when developers push their changes to feature branches in the GitHub repository. While the CI pipeline automatically triggers to validate these changes, merging into the main branch is a controlled, manual process through Pull Requests. This approach allows for code review, conflict resolution, and ensuring all tests pass before changes reach the main branch. Once approved and merged, the CI pipeline triggers again on the main branch to ensure the integrated code remains stable.
- **Build Process:** The build process encompasses compiling the .NET backend application along with building the Angular frontend components. This stage creates all necessary artifacts required for testing and eventual deployment. The process verifies that all components can be successfully built together, catching any compilation or dependency issues early.
- **Automated Testing:** Once the build succeeds, the system initiates a comprehensive testing phase. This includes running unit tests with XUnit framework and executing integration tests to validate component interactions. The process also validates any database migrations to ensure data integrity. Tests serve as quality gates - only passing builds proceed to the next stage, with failed builds triggering immediate developer notifications.

Continuous Deployment Strategy (CD)

This section outlines our approach to deploying the containerized Student Management System in a local Kubernetes environment. The strategy focuses on maintaining system availability and data integrity throughout the deployment process, while leveraging containerization benefits for consistent environments across development and production.

Containerization Overview

Containerization packages an application and its dependencies into a standardized unit (container) that can run consistently across any environment. Using Docker for containerization offers several benefits:

- Consistent environments from development to deployment
- Isolated runtime environments for each component
- Easy scaling and version management
- Efficient resource utilization

Kubernetes Orchestration

Kubernetes serves as our container orchestration platform, managing the deployment and operation of our containerized applications. It provides:

- Automated container management
- Service discovery and load balancing
- Self-healing capabilities (automatic restarts, replacements)
- Scalability and resource optimization

In Kubernetes, a Pod is the smallest deployable unit, representing one or more containers that should be controlled as a single application. Pods in our system run individual components (frontend, backend, or database) with their specific configurations and resource allocations.

Student Management System Deployment Approach

Our continuous deployment strategy involves three main components, with specific communication patterns:

1. **Frontend to Backend Communication:** The Frontend Pod serves the Angular application to users as the primary interface. Through the Frontend Service, which acts as the entry point, external users can access the application. The Frontend communicates with the Backend Pod via REST API calls, which are routed through the Backend Service, ensuring a clean separation of concerns and maintainable service discovery.
2. **Backend to Database Communication:** The Backend Pod houses our .NET WebAPI and manages all business logic and data operations. It establishes connection with the Database Pod through the Database Service, using a secure internal communication channel. This connection remains within the cluster's network, ensuring data access is properly encapsulated and secured from external access.
3. **Data Persistence:** The Database Pod running SQL Server maintains data persistence through a dedicated Persistent Volume, ensuring data survives across pod restarts and redeployments. This pod operates exclusively within the cluster's internal network, with database access strictly controlled and limited to authorized internal communications from the Backend Service.

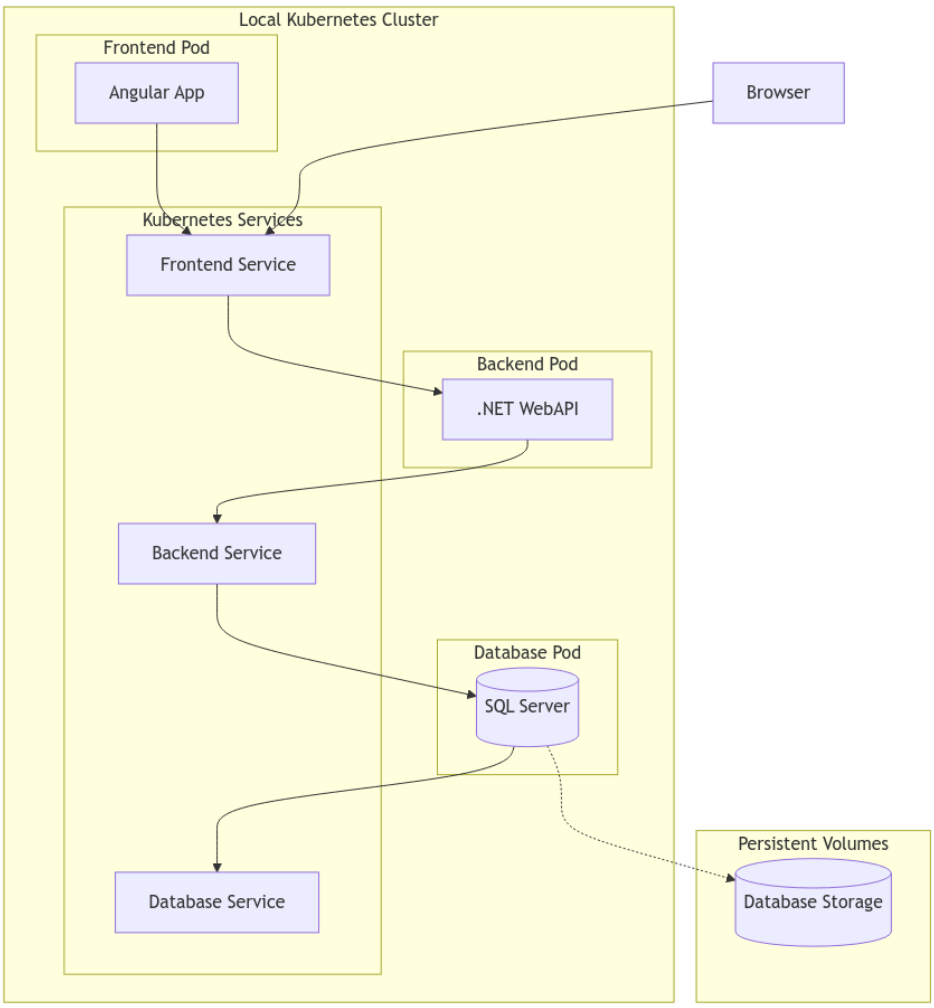

Figure 6- Deployment Architecture: Communication Flow Between Frontend, Backend, and Database Pods

Conclusion

This design document has presented a comprehensive approach to implementing a CI/CD pipeline using a Student Management System as a demonstration platform. The design decisions reflect a balance between simplicity in the demonstration application and robustness in the CI/CD implementation.

Our architectural choices, from Clean Architecture to containerized deployment, ensure that the system remains maintainable and scalable. The CI/CD pipeline design, with its automated

testing, containerization, and deployment stages, demonstrates modern DevOps practices while maintaining manual control over critical integration points.

The design satisfies both the fundamental requirements of a functional Student Management System and the more complex requirements of a production-grade CI/CD pipeline. This approach provides a practical example of how modern software development and deployment practices can be implemented effectively, even in relatively simple applications.

References

Martin, R. C. (2017). *Clean architecture: A craftsman's guide to software structure and design* (1st ed.). Pearson.

Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional.