2025

Continuous & Agile Software Engineering

ANASTASIA TSATSOU

Introduction

Project Objectives

The primary objective of this assessment was to develop and deploy a microservices-based expense tracker application that demonstrates understanding of:

- Microservices architecture design principles
- Git version control and branching strategies
- Database schema design and management
- Containerization technologies
- Continuous Integration and Continuous Deployment (CI/CD) practices
- Application monitoring and observability

Application Overview

The expense tracker application allows users to manage their personal expenses through a RESTful API interface. Users can perform CRUD operations on expense records, categorize expenses (e.g., Food, Entertainment, Transportation), and retrieve filtered expense lists by category.

System Architecture and Design

Microservices Architecture

The application follows a microservices-based two-tier architecture:

Frontend Tier:

- Python FastAPI REST API service
- Handles HTTP requests and business logic
- Provides endpoints for expense management operations
- Implements data validation and error handling

Data Tier:

- MariaDB relational database
- Stores expense records and category information

- Ensures data persistence and integrity
- Supports concurrent access through proper transaction management

Design Rationale

The microservices approach was chosen for several key reasons that align with modern cloud-native development practices. The architecture enables independent scaling of services based on demand, allowing for optimal resource utilization and cost efficiency. Clear separation of concerns facilitates easier maintenance, updates, and debugging by isolating functionality within dedicated services. This approach also supports technology diversity, permitting different services to utilize the most appropriate technology stacks for their specific requirements. Fault isolation ensures that failures in one service do not necessarily cascade to impact other services, improving overall system resilience. Additionally, the microservices pattern enhances development efficiency by enabling teams to work on different services independently, accelerating the development lifecycle.

API Design

The REST API follows RESTful principles with core endpoints that provide comprehensive expense management functionality. The GET endpoints allow retrieval of all expenses or filtered results by category, while POST endpoints enable creation of new expense records. PUT and DELETE endpoints provide update and removal capabilities for existing expenses. Additionally, a dedicated metrics endpoint supports application monitoring and observability requirements.

Database Schema Design

The database schema was designed to support the expense tracking requirements efficiently with one table. The expenses table contains the core expense records including amount, description, category, expense date, and audit timestamps.

Git Workflow and Version Control

Repository Structure

The project repository follows a clear organizational structure:

```
athtech-case/

 ├── app/               # Application source code

|   ├── models/         # Database models

|   ├── routes/        # API route handlers
|   ├── utils/          # Utility functions
|  └── requirements.txt   # Python dependencies
├── database/           # Database-related files
|   ├── init.sql        # Database initialization
|  └── schema.sql       # Database schema
├── docker/             # Docker configuration
|   ├── Dockerfile      # Application container
|  └── docker-compose.yml # Multi-container setup
├── .github/            # GitHub Actions workflows
|  └── workflows/       # CI/CD pipeline definitions
├── tests/              # Application tests
├── docs/               # Project documentation
└── README.md           # Project overview
```

Branching Strategy

The project implements a Git Flow branching model that ensures code quality and deployment stability. The main branch contains production-ready code and is protected with branch protection rules requiring pull request reviews for all changes. The development branch serves as an integration point for new features, where regular testing and validation occurs before promotion to production. Feature branches are short-lived and named descriptively to reflect their specific functionality, such as expense-api or monitoring implementations. These branches merge via pull requests with mandatory code review processes.

Containerization Implementation

Docker Strategy

The application uses multi-container Docker architecture with specialized containers for different responsibilities. The application container handles the Python FastAPI and business logic, while the database container manages data persistence with MariaDB. This separation enables independent scaling, maintenance, and deployment of each component while maintaining clear architectural boundaries.

CI/CD Pipeline Implementation

Continuous Integration Workflow

The CI pipeline automates code validation and container building through comprehensive automation triggered by specific events. Pipeline execution occurs on pushes to main, pull request creation and updates, and manual workflow dispatch when needed. The process begins with code checkout to retrieve the latest repository state, followed by environment setup with Python dependencies. Unit testing executes the complete test suite using pytest to validate functionality. Container building creates Docker images for both application and database components, followed by image testing to validate container functionality.

GitHub Actions Configuration

The workflow definition includes comprehensive automation with triggers on push to main, pull request events, and manual dispatch options. The pipeline consists of two main jobs: a test job that sets up Python environment and runs the test suite, and a build job that creates Docker images and validates container functionality. Each job includes proper dependency management and error handling to ensure reliable pipeline execution.

Testing Implementation

Testing Strategy

The application implements a comprehensive testing approach that covers multiple layers of functionality. Unit tests focus on individual functions and methods while mocking external dependencies such as database connections and APIs to achieve high code coverage exceeding 85%. Integration tests validate API endpoints through end-to-end scenarios, test database interactions, and verify container functionality. API tests specifically validate REST endpoint behavior, test request and response formats, and verify error handling scenarios to ensure robust application behavior under various conditions.

Monitoring and Observability

Application Metrics

The application exposes monitoring metrics through a dedicated endpoint that provides comprehensive operational visibility. Metrics collected include request count and response times for performance monitoring, database query count for resource management, memory and CPU usage for capacity planning.

Conclusion

This project successfully demonstrates the implementation of a modern, cloud-native microservices application following industry best practices. The expense tracker application meets all specified requirements while incorporating advanced features for maintainability, scalability, and observability.

Key Achievements

1. **Successful Microservices Implementation:** Clean separation of concerns with REST API and database tiers
2. **Comprehensive CI/CD Pipeline:** Automated testing, and building processes
3. **Effective Containerization:** Optimized Docker containers with proper security practices
4. **Robust Testing Strategy:** Comprehensive test coverage with automated execution
5. **Production-Ready Monitoring:** Health checks and metrics collection for operational excellence