Zewail City of Science, Technology and Innovation
University of Science and Technology
School of Computational Sciences and Artificial Intelligence

CSAI 203 - Fall 2025
Introduction to Software Engineering
|
Software Design Document
for
Adaptive Collaborative Code Learning Lab (ACCL)

Team Number: ***Team 18***
Team Members:
Amr Yasser 202301043
Omar Hazem Ahmed 202300800
Omar Darwish 202301146
Abdelrahman Mohamed 202301645
Hady Emad Saeed 202301707
Representative ID: ***202301043***
Representative Contact info:
s-amr.anwar@zewailcity.edu.eg

Version: 1.0
Organization: Zewail City
Date: ***20/11/2025***

1. Introduction

1.1 Purpose

The purpose of this Software Design Document (SWDD) is to define how the Adaptive Collaborative Code Learning Lab (ACCL) system will be structured and implemented. It provides a detailed description of the system's architecture, components, data design, and key interactions to ensure that the development team can build the system accurately according to the defined requirements.

Terminology Note:
This document uses the abbreviations defined in the ACCL SRS Section 1.3. Full terms (e.g., ACCL, UI, API, MVC) are not redefined here for brevity.

1.2 Scope of the Design Phase

This design phase covers the structural and behavioral design necessary to implement the Phase-4 prototype of the ACCL system. It defines the system architecture, module organization using the Model–View–Controller (MVC) pattern, Flask Blueprint structure, database (DB) schema using SQLAlchemy, essential UI views, sequence and class diagrams for core workflows, and operational interactions such as code execution in the sandbox environment. Production-level scalability and advanced AI model customization are out of scope for this phase.

1.3 Intended Audience

This document is intended for the ACCL development team (implementers and integrators), course instructors and TAs (for evaluation and verification), QA and test engineers (to derive test cases), DevOps/demonstrators (to deploy and run the prototype), the team lead/project manager (for planning and progress tracking), and future maintainers or auditors who need to understand the system design. End users (students/instructors) may consult selective high-level sections for orientation, but the primary audience is technical staff and evaluators.

1.4 Overview of the Contents

This document provides the design foundation for ACCL and is organized into five major sections. Section 1 introduces the document purpose, scope, and organization. Section

2 gives a high-level system overview including its main features and key design goals. Section 3 presents the architectural design with diagrams, component descriptions, and the selected architecture style. Section 4 provides the detailed design including the MVC structure, diagrams, UI layout, data design, and component responsibilities. Section 5 concludes with a summary of major design decisions.

2. System Overview

2.1 Brief Description of the System

The **Adaptive Collaborative Code Learning Lab (ACCL)** is a web-based platform that enables instructors to publish programming assignments and test cases, and allows students to submit, run, and iterate on code in an isolated sandbox. The system automates grading by executing instructor-provided tests, records per-test results, generates progressive AI-assisted hints using pre-trained models, and maintains versioned submission history with diffs. Instructor features include similarity detection (plagiarism advisories), peer review workflows, dashboards for analytics, and CSV export for grading.

ACCL follows a modular, service-oriented design: a web application (Flask with Blueprints) implements the MVC layers and user-facing UI, a background worker system (Redis/queue) manages sandbox jobs that run in Docker containers, and a persistence layer (SQLAlchemy-backed database and object storage) stores submissions, test results, hints, and audit logs. An AI service wrapper provides cached hint generation and an extensible similarity engine supports token/embedding hybrid comparisons; repositories and service layers encapsulate business logic for maintainability and testability.

2.2 Key Design Goals and Constraints

Design goals

- **Correctness & clarity:** reliable, repeatable grading with concise per-test results and progressive, non-solution AI hints.
- **Safety:** run untrusted code in isolated, resource-limited sandboxes (no network, timeouts, user remapping).
- **Modularity & maintainability:** MVC + Blueprints, Repository/Service layers to enable testability and pluggable components (AI, similarity, storage).

- **Usability & auditability:** clear student/instructor flows, versioned submissions, similarity highlights, and persistent audit logs.

- **Performance targets (prototype):** per-test timeout ≤ 5s; median grading + deterministic hint ≤ 4s (AI path target ≤ 6s).

Constraints

- **Tech stack (Phase-4):** Flask, SQLAlchemy/SQLite, Redis/queue, Docker sandboxes — chosen for rapid development but with known limits.

- **Concurrency & testing:** SQLite has limited concurrent writers; use WAL and file-backed DB for multi-process integration tests

- **AI policy:** only pre-trained models allowed; implement deterministic fallback when external providers are unavailable.

- **Operational limits:** demo/CI resources are constrained; wireframes must use draw.io/PPT (no Figma or auto-code tools).

3. Architectural Design

3.1 System Architecture Diagram

View: System Architecture Diagram

3.2 Discussion of Architectural Style and Components

The ACCL system adopts a **monolithic, layered architecture** with an internal **MVC/MVT** organization implemented in a single Flask application. This approach minimizes deployment complexity for Phase-4 while preserving clear separation of concerns (presentation, application logic, domain services, persistence).

Key components

- **Web application (Flask, MVC/MVT):** Jinja2/Bootstrap views, controllers/routes for request handling, and services/repositories for business logic and data access.

- **Data & storage:** SQLAlchemy/SQLite for relational data, Redis for cache and job queue, local file/object storage for artifacts and exports.
- **Background worker & queue:** Separate process (RQ/Celery) that consumes jobs, orchestrates sandbox runs, and persists results.
- **Sandbox host (Docker):** Ephemeral, isolated containers (no network, resource quotas, timeouts) for safe code execution.
- **External adapters:** Pluggable AI and embedding providers accessed via service wrappers with caching and deterministic fallback.
- **Observability:** Audit logs, test-run artifacts, and basic metrics persisted for verification and debugging.

Rationale: Monolithic + MVC enables rapid, testable development and straightforward grading/demonstration; companion worker and sandbox isolate risky execution without requiring a microservices deployment.

3.3 Technology Stack and Tools

Backend & Core Application

- Python 3.x
- Flask (routing, controllers, templating)
- SQLAlchemy ORM
- Redis (caching, job queue)
- RQ or Celery (background workers)
- Docker (sandboxed execution)

Frontend

- Jinja2 templates

- Bootstrap 5
- Vanilla JS + AJAX for interactive views

Data & Storage

- SQLite (relational database)
- Local file/object storage for submissions, results, and exports

AI & Analysis

- External AI provider API (hints, explanations)
- Embedding service API (similarity scoring)

Development & Testing Tools

- Git/GitHub
- pytest + coverage
- flake8/black (linting and formatting)
- Postman or cURL (API testing)
- Docker Compose (local orchestration)
- GitHub Actions (CI/CD)

4. Detailed Design

4.1 Model–View–Controller (MVC) Design Pattern

4.1.1 Description of MVC Pattern

- **Model:** encapsulates the domain data and business rules (SQLAlchemy models, repositories, domain services). Models are responsible for persistence, validation, and core operations on application data.

- **View:** presents data to the user (Jinja2 templates, HTML fragments, client-side scripts). Views are passive renderers and contain no business logic.
- **Controller:** handles HTTP requests, performs input validation, coordinates use-case flows, invokes services/repositories, and selects views or responses.

Why MVC was chosen for ACCL

- **Separation of concerns:** MVC cleanly isolates presentation, request handling, and domain logic, reducing coupling and simplifying maintenance.
- **Testability:** Controllers remain thin and delegate business logic to services/repositories and models, making unit tests straightforward and enabling lightweight mocks for integration tests.
- **Clarity for grading and team collaboration:** The explicit roles (models, views, controllers) make responsibilities easy to document and assess—important for course evaluation and for multiple team members working across layers.
- **Fits Flask and course constraints:** Flask's MVT/MVC-style architecture (Jinja2 views, Blueprints for controllers, SQLAlchemy models) maps directly to MVC, enabling rapid implementation while preserving the design patterns required for bonus points (factory, repository, singleton).
- **Extensibility:** MVC facilitates plugging alternative components (e.g., swapping AI adapters or moving persistence to another DB) with minimal changes to presentation or routing logic.

4.1.2 Mapping of Project Components to MVC

- **Model Layer:**

    - **User, Assignment, Submission, Result** SQLAlchemy models
    - **UserRepository, SubmissionRepository, ResultRepository**
- **View Layer:**
    - Jinja2 templates for student dashboard, assignment pages, submission results, instructor dashboard
    - Static assets (CSS/JS) for UI styling and interactivity
- **Controller Layer:**
    - **Flask Blueprints: auth, assignments, submissions, instructor**
    - Route handlers managing requests, invoking services, and returning rendered views
- **Service/Repository Layer (Business Logic):**
    - Sandbox execution manager, similarity detection, grading engine
    - Handles background jobs and communicates with Model and external AI/embedding services

4.1.3 Responsibilities of Model, View, and Controller

- **Model:**
    - Manages system data and business rules
    - Handles database interactions via SQLAlchemy models and repository classes
    - Ensures data integrity and persistence
- **View:**
    - Presents data to users via Jinja2 templates and static assets

    - Provides user interface for students, instructors, and admins
    - Reflects real-time updates from Model and Controller
- **Controller:**
    - Handles HTTP requests and input from users
    - Coordinates between Model and View
    - Invokes business logic, background jobs, and external AI services
    - Ensures correct workflow and data flow throughout the system

4.1.4 Interaction Between Components

In ACCL, the MVC components interact as follows:

1. **User Request:** A student, instructor, or admin interacts with the View (UI layer).
2. **Controller Handling:** The Controller receives the request, validates input, and determines the appropriate business logic.
3. **Business Logic & Data Access:** The Controller calls Model methods via repositories or services to read/update data.
4. **Response Preparation:** The Model returns data to the Controller, which prepares it for presentation.
5. **View Rendering:** The Controller passes the data to the View, which renders HTML (via Jinja2) and delivers it to the user.
6. **Asynchronous Tasks:** For long-running operations (e.g., sandbox execution or AI analysis), the Controller enqueues tasks in the Worker, which interacts with Model and storage, updating results asynchronously.

4.2 UML Diagrams

4.2.2 Detailed Class Diagram

View: UML Class Diagram

4.2.3 Sequence Diagram

View Sequence Diagrams:
- Submit Assignment
- Manual Regrade
- Batch Plagiarism Detection
- AI Hint Request
- Peer Review Submission
- Student Course Enrollment
- MVC Interaction

4.3 UI/UX Design

4.3.1 Wireframes / Mockups

View UI/UX Wireframes:

Assignment Dashboard:
- [Desktop v1] [Desktop v2]
- [Mobile v1] [Mobile v2]

Code Editor:

- Desktop: [v1] [v2] [v3] [v4]
- Mobile: [v1] [v2]
- Merged: [v1] [v2] [v3] [v4]

Progress Feedback:

- [Desktop v1] [Desktop v2]
- [Mobile v1] [Mobile v2]

Similarity Dashboard:

- [Desktop v1] [Desktop v2]
- [Mobile v1] [Mobile v2]

Submission History:

- [Desktop v1] [Desktop v2]
- [Mobile v1] [Mobile v2]

4.4 Data Design

4.4.1 Database Schema / ER Diagram

View: ER Diagram

4.4.2 File Structure / Data Storage Model

The system uses a structured local filesystem storage area for Phase-4 to persist submission files, sandbox run outputs, exported grading reports, and periodic backups. The web application and background worker interact with this storage through service-layer abstractions, while associated metadata (e.g., file path, owner, size, timestamps) is stored in the database for consistency and traceability.

A corresponding diagram in this section illustrates the directory organization and how different components read or write to each storage location.

4.4.3 Data Dictionary

See: Data Dictionary

5. Conclusion

5.1 Summary of Design Phase

This design phase produced a complete, implementation-ready architecture for ACCL: a monolithic, layered Flask application organized with MVC/MVT, Blueprints, an App Factory, and Repository/Service abstractions. Key deliverables include the System

Architecture diagram, component and data-flow diagrams, five sequence diagrams, detailed class/domain models, and wireframes that together specify presentation, control, business, and persistence responsibilities.

Major design decisions: use a monolithic deployable for Phase-4 to simplify development and grading; isolate code execution in ephemeral Docker sandboxes managed by a background worker (Redis + RQ/Celery) to protect the web process; adopt SQLAlchemy/SQLite for Phase-4 persistence with clear notes on concurrency testing; and provide pluggable adapters for AI and embedding providers with caching and deterministic fallbacks. The design explicitly implements Factory, Repository, and Singleton (extensions) patterns and documents these for bonus verification.

The design emphasizes testability and observability: controllers are thin, business rules live in services/repositories, integration points are adapter-wrapped, and audit logs plus persisted test artifacts support verification. CI, linting, and pytest-based unit/integration test plans are included to validate core flows (submit → queue → sandbox → grade → feedback) before Phase-4 delivery (Core Functionality Prototype).
