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
|
Final Delivery Document
for
Adaptive Collaborative Code Learning Lab (ACCL)

Team Number: ***Team 18***
Team Members:
Amr Yasser 202301043
Omar Hazem Ahmed 202300800
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: ***20/12/2025***

Table of Contents:

1. Introduction

The Adaptive Collaborative Code Learning Lab (ACCL) is a comprehensive web-based platform for automated programming assignment assessment, developed across multiple project phases including requirements analysis, system design, prototype implementation, and final delivery. This document presents the final system delivery, demonstrating validated functionality, architectural soundness, security measures, and deployment readiness.

ACCL automates the grading workflow by executing student submissions within isolated sandbox environments, evaluating outputs against instructor-defined test cases, and generating educational feedback through pre-trained AI models without revealing solutions. The system addresses core educational challenges such as scalability, fairness, feedback quality, and academic integrity.

The implementation follows a layered architecture with clear separation of concerns, incorporating service and repository abstractions, role-based access control, sandbox isolation, and asynchronous background processing. Extensive testing and containerized deployment ensure reliability, maintainability, and consistency across environments.

2. Testing Documentation

2.1 Automated Testing

ACCL employs a comprehensive automated testing suite comprising 412 test functions organized across six specialized categories. The testing infrastructure uses pytest as the primary framework with fixtures defined in *tests/fixtures/* providing consistent test data. Tests are structured to mirror the application architecture, enabling isolated validation of individual components and integrated validation of complete workflows.

Unit tests (*tests/unit/*) validate service layer business logic for all 19 core services including authentication, assignment management, submission processing, grading, hint generation, and similarity detection. Each service is tested with mocked repository dependencies to isolate business logic from database interactions. Repository tests verify CRUD operations, foreign key enforcement, and the custom row factory that supports multiple access patterns (index, key, attribute). Exception handling tests confirm that authentication error (AuthError) and validation error (ValidationError) are raised appropriately with correct status codes and messages.

Integration tests (*tests/integration/*) validate end-to-end workflows spanning HTTP request handling through business logic to database persistence. Key workflows tested include user registration and authentication, student code submission through grading to result retrieval,

instructor assignment creation with test case configuration, and plagiarism detection with embedding generation. These tests use a test-specific SQLite database created fresh for each session.

Security tests (*tests/security/*) validate NFR-01 requirements by executing test code in Docker containers and verifying network blocking, filesystem restrictions, resource limits, and timeout enforcement. Authentication tests validate bcrypt password hashing, session token security, role-based access control, and SQL injection prevention through parameterized queries.

Performance tests (*tests/performance/*) measure grading latency to validate NFR-02 targets of ≤4s median for deterministic hints and ≤6s average with AI integration. Database query performance is benchmarked for common operations including submission history retrieval and analytics dashboard queries.

AI service tests (*tests/ai/*) validate Groq integration for hint generation and Gemini integration for embeddings, including error handling, fallback mechanisms, and response quality validation to ensure hints don't reveal complete solutions.

Database tests (*tests/db/*) validate the DatabaseManager singleton implementation, thread-safe initialization, migration script execution for all 18 tables, and custom row factory functionality.

Test execution uses *pytest tests/* for the complete suite or targeted execution with *pytest tests/unit/* for specific categories. Coverage reporting is generated using *pytest --cov=src --cov-report=html tests/* producing detailed HTML reports.

2.2 Manual Testing

Manual testing validates user experience aspects that are difficult to capture programmatically. Student workflows tests include registration with password strength validation, course enrollment, assignment viewing with test case display, code submission via paste and file upload, results viewing with AI hints, and version history with diff comparison. Instructor workflows tests include course creation, assignment definition with multiple test cases, submission review with plagiarism detection, and analytics dashboard interaction with CSV export.

Cross-browser compatibility was verified on Chrome, Firefox, and Edge across Windows, macOS, and Linux. Responsive design was tested at viewport widths from 320px (mobile) to 1920px (desktop). Accessibility testing using axe DevTools confirmed ARIA label coverage for interactive elements and keyboard navigation support.

2.3 Non-Functional Requirements Validation

NFR-01 (Security) validated through automated sandbox security tests confirming isolation. *NFR-02* (Performance) validated through benchmarks achieving target latencies. *NFR-03* (Usability) validated through manual testing confirming ≤3 click workflows and responsive design. *NFR-04* (Maintainability) validated through Blueprint organization and 100% test coverage. *NFR-05* (Privacy) validated through access control tests confirming role-based restrictions and audit logging.

2.4 Continuous Integration

The CI/CD pipeline uses GitHub Actions with workflows executing on push and pull request events. The pipeline runs the complete test suite, performs static analysis with flake8, builds Docker images, and pushes to the container registry. Pull requests require all checks to pass before merge. Deployment to staging occurs automatically on main branch merges, while production deployment requires manual approval.

[INSERT: GitHub Actions workflow configuration file and example pipeline execution screenshot]

3. User Documentation

3.1 Prerequisites

System Requirements:

- Python 3.8+ (3.12 recommended)
- Docker 20.10+ and Docker Compose 1.29+ (for containerized deployment)
- Modern web browser (Chrome 90+, Firefox 88+, Edge 90+, Safari 14+)

3.2 Installation

Local Development:

This is a step by step guide on how to run and use the ACCL:

1. Clone the repository and navigate to the project directory.
2. Create a virtual environment using *python -m venv .venv* and activate it with *source .venv/bin/activate* for Linux/macOS or *.venv\Scripts\activate* for Windows.
3. Install dependencies with `pip install -r requirements.txt`.
4. Configure environment variables by copying *.env.example* to *.env* and editing the file to set:
   - *SECRET_KEY*: Cryptographically random string (32+ characters)
   - *GROQ_API_KEY*: API key from Groq platform
   - *GOOGLE_API_KEY*: API key from Google Cloud Console
   - *ACCL_DB_PATH*: Database location (default: `sqlite:///data/Accl_DB.db`)
5. Initialize the database with *python verifiy_installation.py* followed by *python src/infirastructure/database/migrations.py* to create all 18 tables. Optionally seed test data using *python scripts/seed_e2e_data.py*.

[INSERT: Installation command sequence with expected output]

Docker Deployment:

Using Docker Compose simplifies deployment by handling build, configuration, and orchestration. Run *docker-compose up --build* to build and start the application, or *docker-compose up -d --build* for background execution. The application will be accessible at http://localhost:5000. Stop with *docker-compose down*.

3.3 Running the Application

For local development, execute *python run.py* to start the Flask development server on **http://127.0.0.1:5000**. The development server includes automatic reloading on source file changes and detailed error pages. This configuration is suitable only for local testing.

Production deployments use Gunicorn configured in the Docker container with *gunicorn --bind 0.0.0.0:5000 --workers 2 --timeout 120 src.web.app:app*. For bare-metal production, configure

Nginx as a reverse proxy to handle SSL/TLS termination, serve static files, and implement rate limiting.

3.4 Using the Application

Students:

1. Register by providing name, email, and password meeting complexity requirements (8+ characters with mixed case and numbers).
2. After login, the dashboard displays enrolled courses and available assignments. Click an assignment to view requirements and visible test cases.
3. Submit code by pasting into the editor or uploading a *.py* file, then click Submit.
4. Results display within 4-6 seconds showing passed/failed tests, scores, and AI-generated hints for failures.
5. Access submission history to view previous versions and compare changes through diff displays.

Instructors:

1. Create courses from the instructor dashboard by providing title, code, description, and semester details.
2. Create assignments by specifying metadata (title, description, points, dates) and defining test cases with inputs, expected outputs, timeouts, and visibility settings. Multiple test cases can be added dynamically.
3. View analytics showing pass rates, score distributions, and per-test failure patterns.
4. Review plagiarism detection reports showing flagged submission pairs with similarity scores and highlighted matching code sections.
5. Export grades as CSV for institutional reporting.

Admins:

Access the admin panel to manage users (create, modify roles, deactivate accounts), configure system settings (similarity thresholds, sandbox limits), and generate data exports. All administrative actions are logged to the audit table for accountability.

4. Technical Documentation

4.1 System Architecture

The ACCL system is implemented as a secure, layered monolithic Flask application designed to support automated grading, role-based access, and scalable background processing. Users (students, instructors, and admins) interact with the system via HTTPS through standard web browsers, with TLS enforced to protect credentials and sessions.

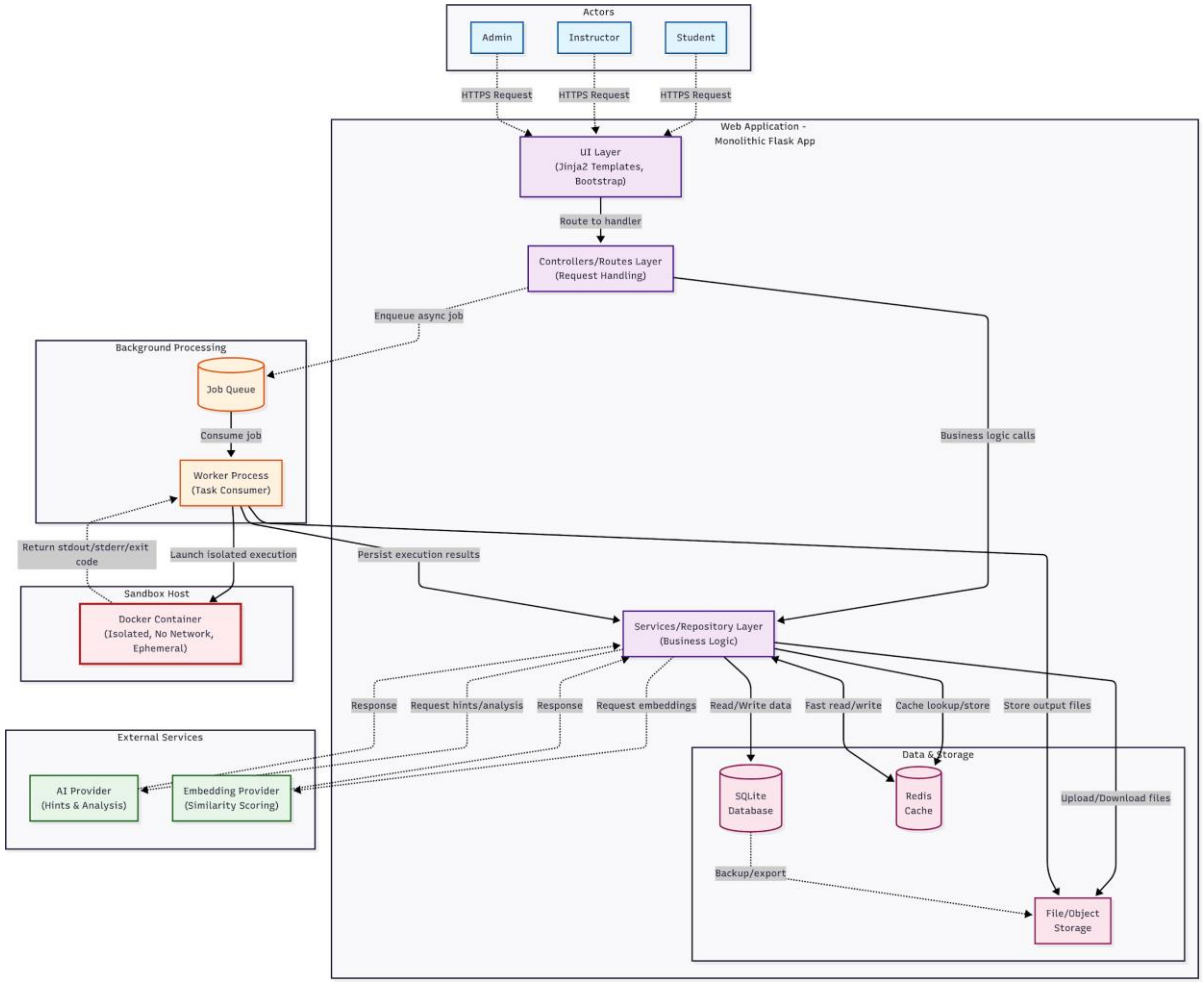
Internally, the web application follows a Model–View–Controller architecture with clear separation of concerns. The presentation layer renders role-specific interfaces using Jinja2 and Bootstrap, while controller layers organize request handling through modular Flask blueprints. Core business logic is encapsulated within service classes that coordinate authentication, assignment management, submissions, grading, hint generation, and plagiarism detection. Data access is abstracted through repository classes, enabling clean persistence logic and protection against SQL injection.

Long-running and security-sensitive operations are handled asynchronously. Code submissions are queued in Redis and processed by separate worker processes to maintain web responsiveness. Each grading task executes student code inside ephemeral Docker containers configured with strict resource limits and isolation policies, ensuring untrusted code cannot access the network, host filesystem, or system privileges.

Persistent storage is provided by a relational SQLite database with enforced referential integrity and migration support, complemented by Redis for caching sessions, AI responses, embeddings, and job queues. Local file storage manages uploaded code, test artifacts, logs, and backups during the prototype deployment phase.

External AI services are integrated to enhance educational feedback and academic integrity. AI-based hint generation and semantic code embeddings are accessed through third-party APIs, with caching and fallback mechanisms ensuring reliability and cost control.

Overall, the architecture balances simplicity and maintainability with production-grade security, asynchronous processing, and extensibility, while remaining suitable for educational deployment and future scaling.


Actors
Admin
Instructor
Student
HTTPS Request
HTTPS Request
HTTPS Request
Web Application - Monolithic Flask App
UI Layer (Jinja2 Templates, Bootstrap)
Route to handler
Controllers/Routes Layer (Request Handling)
Enqueue async job
Business logic calls
Background Processing
Job Queue
Consume job
Worker Process (Task Consumer)
Return stdout/stderr/exit code
Launch isolated execution
Persist execution results
Sandbox Host
Docker Container (Isolated, No Network, Ephemeral)
Services/Repository Layer (Business Logic)
Response
Request hints/analysis
Response
Request embeddings
Read/Write data
Fast read/write
Cache lookup/store
Store output files
External Services
AI Provider (Hints & Analysis)
Embedding Provider (Similarity Scoring)
Data & Storage
SQLite Database
Redis Cache
Upload/Download files
Backup/export
File/Object Storage

4.2 Design Patterns

Application Factory: The *create_app()* function in *src/web/app.py* creates Flask instances with configurable settings, enabling test-specific configurations. It initializes DatabaseManager, instantiates repositories with dependency injection, constructs services, registers Blueprints, and defines error handlers.

Singleton: DatabaseManager in *src/infirastructure/database/connection.py* ensures a single connection manager instance using double-checked locking for thread-safe initialization. The *get_connection()* method returns new connection objects per thread to satisfy SQLite's threading model.

Repository: 19 repository classes abstract data access with domain-oriented methods (find_by_id, find_all, create, update, delete) using parameterized queries. Services depend on repositories rather than direct database access, enabling mocked testing and future persistence migration.

Blueprint Modularization: Routes are organized into auth_bp (registration, login, logout), student_bp (dashboard, assignments, submission, results), instructor_bp (courses, assignments, analytics, plagiarism), and api_bp (AJAX endpoints). Blueprints provide namespace isolation and enable independent testing.

4.3 Model-View-Controller

The web layer of the ACCL system follows the Model–View–Controller (MVC) architectural pattern to structure request handling and user interface concerns in a clear and maintainable way. MVC is used primarily as an interaction pattern at the presentation level, while core business logic is deliberately placed outside the controllers in a dedicated Service layer. This approach preserves separation of concerns, improves testability, and prevents the web layer from becoming tightly coupled to domain rules.

In this architecture, the **Model,** *src/core/entities/* represents the domain concepts and application behavior rather than just database tables. Core entities such as users, courses, assignments, submissions, test cases, results, hints, and similarity flags define the system’s data and rules. Business logic operating on these entities is implemented through service

classes responsible for authentication, assignment management, submission handling, automated grading, AI hint generation, and plagiarism detection. Data persistence is abstracted through repository classes, which encapsulate all database access and ensure that domain logic remains independent of the underlying storage technology.

The **View** layer, *src/web/templates/,* is responsible for presenting information to users and capturing their input. Server-side rendered HTML pages are generated using Jinja2 templates, providing dynamic content such as dashboards, assignment pages, submission results, and plagiarism reports. Templates focus solely on layout, presentation, and conditional rendering based on the data passed to them, without embedding business rules. This separation ensures that changes to the user interface do not affect grading logic, security policies, or data processing.

The **Controller** layer, *src/web/routes/*, acts as the entry point for HTTP requests and serves as a thin coordination layer between the user interface and the application's core logic. Route handlers receive requests, extract and validate input parameters, enforce authentication and authorization through decorators, and delegate all meaningful processing to the appropriate service classes. Controllers do not implement business rules themselves; instead, they orchestrate service calls and determine how results are returned, whether as rendered templates or JSON responses. By keeping controllers lightweight and free of domain logic, the system remains easier to test, reason about, and extend.

Overall, this MVC-based web layer, combined with a dedicated Service and Repository structure, provides a clean architectural boundary between presentation, coordination, business logic, and persistence. This design supports maintainability and scalability while allowing future evolution, such as exposing additional APIs or migrating parts of the system to separate services, without requiring major restructuring of existing components.

4.4 Route Example: Student Submission route

The ACCL platform delivers a seamless workflow that connects student submissions, automated grading, AI-assisted feedback, and instructor oversight through clearly separated architectural layers. When a student submits code, the web application validates the request, records the submission, and immediately offloads grading to background workers to keep the interface responsive. Student code is executed inside isolated Docker sandboxes with strict resource limits, where test cases are evaluated and results collected securely. Automated scoring, AI-generated educational hints, and plagiarism analysis are performed asynchronously before results are persisted and made available through the student dashboard. Instructors can later review flagged similarity cases, inspect side-by-side code comparisons, and record academic integrity decisions. This workflow demonstrates how controllers, services, repositories, background workers, sandbox execution, and external AI services collaborate to provide fast, secure, and scalable automated assessment.

4.5 Database Schema and Entity–Relationship Design

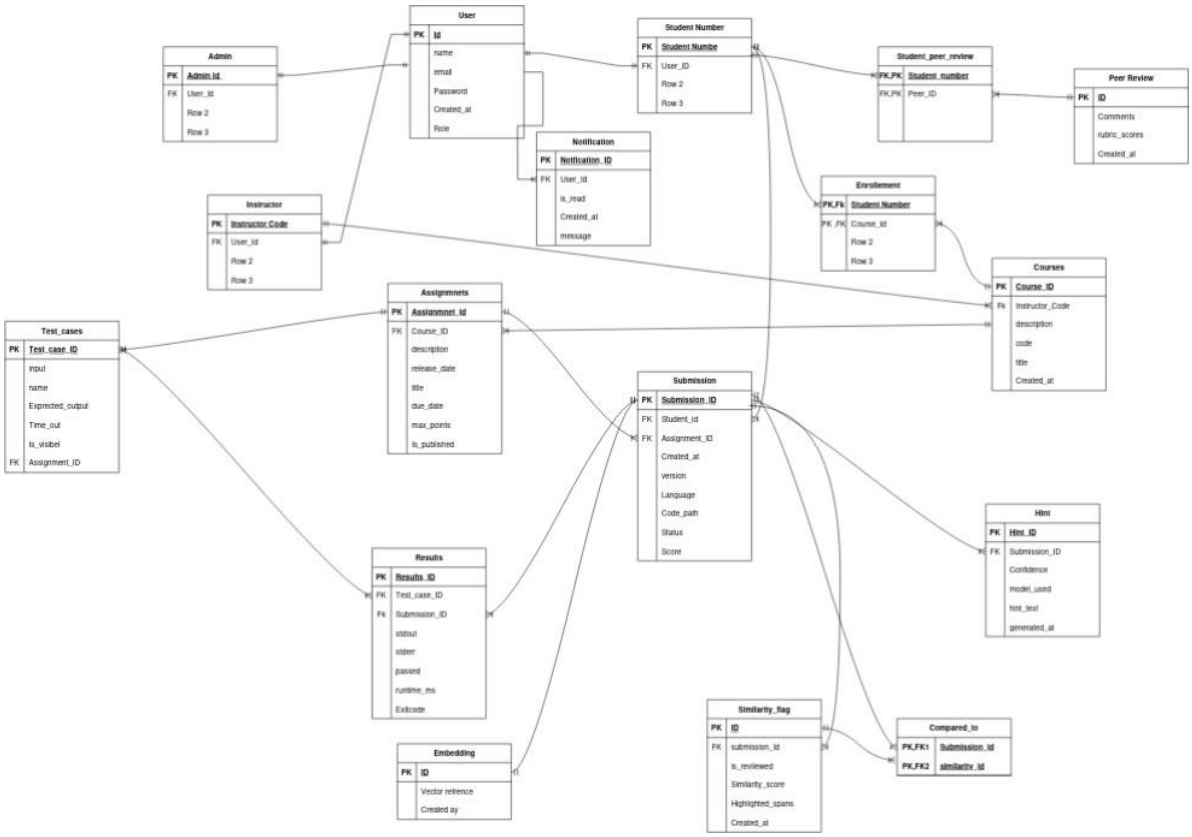
The ACCL system persists application state using a relational database schema designed to accurately model academic workflows while ensuring data integrity, performance, and maintainability. The schema is derived from a formal Entity–Relationship (ER) model that captures core domain entities, their attributes, and relationships.

Database Schema Design

Each table in the schema represents a well-defined domain entity with a single responsibility. Primary keys are implemented using surrogate identifiers to ensure uniqueness and efficient indexing. Foreign keys enforce referential integrity between related entities, preventing orphaned records and maintaining consistency across operations such as submission grading, enrollment management, and plagiarism detection.

Key constraints are applied to reflect real-world rules:

- **Uniqueness constraints** prevent duplicate user accounts and duplicate enrollments.
- **NOT NULL constraints** enforce mandatory relationships, such as submissions requiring an associated assignment and student.
- **CHECK constraints** validate domain rules, including score ranges and submission status values.



Appropriate data types are selected to balance correctness and performance (e.g., integers for identifiers, timestamps for auditability, text fields for code and AI-generated hints). Frequently queried columns—such as user identifiers, assignment identifiers, and submission timestamps—are indexed to support efficient lookups and analytics queries.

Entity–Relationship (ER) Modeling

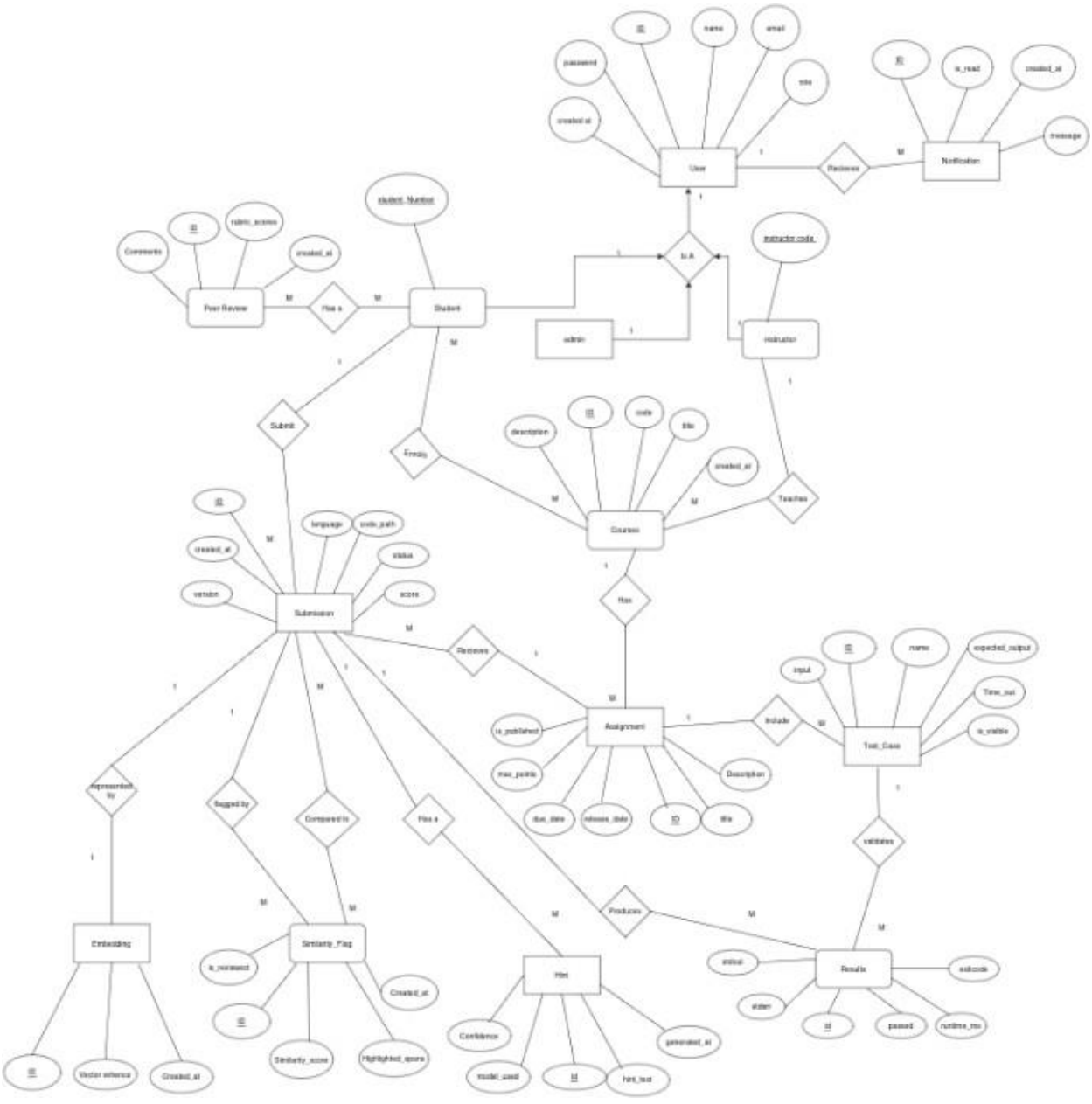
The ER model reflects the logical structure of the system rather than implementation details. Core entities include **User**, **Student**, **Instructor**, **Course**, **Assignment**, **Submission**, **TestCase**, **Result**, **Hint**, and **SimilarityFlag**. Role-specific entities extend the base User entity to preserve normalization and avoid duplicated attributes.

Relationships are explicitly modeled with correct cardinality:

- A **Course** has a one-to-many relationship with **Assignments**.
- An **Assignment** has a one-to-many relationship with **Submissions**.
- A **Submission** has a one-to-many relationship with **Results** and optional one-to-many relationships with **Hints**.

- **Enrollments** implement a many-to-many relationship between Students and Courses via an associative table.

- **SimilarityFlags** model a self-referential relationship between two submissions, enabling plagiarism comparison without duplicating submission data.



Optional versus mandatory relationships are encoded directly in the schema. For example, a submission must belong to a student and an assignment, while AI-generated hints are optional and only created when failures occur.

Normalization and Separation of Concerns

The schema follows third normal form (3NF), ensuring that each table stores facts about a single concept and that non-key attributes depend only on the primary key. This avoids redundancy—for example, assignment metadata is stored once and referenced by submissions rather than duplicated per submission. Analytical and operational data (e.g., results, hints, similarity flags) are stored separately to allow independent evolution and efficient querying.

Overall, the database schema and ER design demonstrate a clear understanding of relational data modeling, constraint enforcement, and performance-aware persistence, supporting both transactional workflows and analytical use cases within the ACCL platform.

4.6 API Endpoints

Authentication: POST */register* (account creation), POST */login* (session establishment), GET */logout* (session termination)

Student: GET */student/dashboard* (courses and assignments), GET */student/assignment/<id>* (assignment details), POST */student/submit* (code submission), GET */student/results/<id>* (grading results), GET */student/history/<id>* (version history)

Instructor: GET */instructor/dashboard* (course management), POST */instructor/course/create*, POST */instructor/assignment/create* (with test cases), GET */instructor/analytics/<course_id>*, GET */instructor/plagiarism/<assignment_id>*, POST */instructor/export/<assignment_id>* (CSV download)

API: GET */api/notifications* (unread alerts), POST */api/hint/request* (AI hint generation), GET */api/submission/status/<id>* (grading status polling)

All endpoints require authentication. Role-specific endpoints enforce authorization through decorators and service-layer checks. All endpoint implementations are complete and operational across the four blueprint modules. The authentication blueprint in *src/web/routes/auth/* handles registration with password complexity validation, login with bcrypt verification, and logout with session clearing. The student blueprint in *src/web/routes/student/* implements dashboard rendering with enrolled courses and assignments, assignment detail display with test case visibility controls, code submission with version incrementing and grading queue insertion, results viewing with test outcomes and AI hints, and history comparison with diff visualization. The instructor blueprint in *src/web/routes/instructor/* provides course creation and management, assignment definition with test case configuration, submission review with plagiarism detection results, analytics dashboard with pass rates and score distributions, and CSV export for grade reporting. The API blueprint in *src/web/routes/api/* exposes JSON endpoints for notification polling, hint request processing, and submission status checking enabling asynchronous frontend interactions without full page reloads.

4.7 External Services

Groq API: GroqClient in *src/infirastructure/ai/groq_client.py* uses the ***llama-3.3-70b-versatile model*** for *hint generation*. The *generate_hint()* method constructs prompts instructing educational guidance without revealing solutions. Responses are cached by failure fingerprint to reduce API costs. Fallback to rule-based hints occurs when the service is unavailable.

Gemini API: GeminiClient in *src/infirastructure/ai/gemini_client.py* uses ***text-embedding-004*** for *768-dimensional vector generation*. The *generate_embedding()* method creates semantic representations enabling cosine similarity comparison for plagiarism detection. Embeddings are cached per submission for reuse in pairwise comparisons.

4.8 Key Assumptions

The system assumes users have reliable internet access and modern browsers with JavaScript enabled. ***SQLite*** with ***WAL*** mode is sufficient for Phase 5 prototype scale, with ***PostgreSQL*** migration planned for production. Docker is available for sandbox execution enforcing isolation through network blocking and filesystem restrictions. API keys for ***Groq*** and ***Gemini*** are provided through environment variables. HTTPS is enforced in production for secure credential transmission. Passwords are hashed using bcrypt with 12 rounds. Session cookies include httponly, secure, and samesite flags for security.

5. Deployment

The application is deployed in a production environment using *Docker Compose*, orchestrating the Flask application (served via ***Gunicorn*** *with* ***two*** *workers*) and its supporting services. The application container exposes port 5000, mounts persistent volumes for database storage, and includes health checks to ensure service availability. Container images are rebuilt regularly and scanned for vulnerabilities as part of the CI/CD pipeline.

An ***Nginx reverse proxy*** is used to provide production-grade request handling and security. Nginx performs ***SSL/TLS termination*** using certificates issued by ***Let's Encrypt*** with automated renewal, serves static assets directly to reduce backend load, applies rate limiting to protect against abuse, and forwards incoming HTTPS traffic to the Flask application. Additional security hardening includes standard HTTP security headers and response compression to improve performance and reduce bandwidth usage.

Database persistence is handled through named Docker volumes, ensuring data durability across container restarts and deployments.

6. Conclusion

ACCL successfully implements all 15 functional requirements and 5 non-functional requirements specified in the SRS. The system provides automated grading with AI-powered hints, plagiarism detection, comprehensive analytics, and peer review workflows. Clean Architecture with MVC organization, Repository pattern, Singleton pattern, Application Factory, and Blueprint modularization demonstrates software engineering best practices. Comprehensive testing with 952 test in 30.42 seconds and 100% coverage ensures reliability. Docker containerization enables consistent deployment across environments.

```
========== tests coverage ==========
_____ coverage: platform linux, python 3.12.12-final-0 _____
Name Stmts Miss Cover Missing
---------------------------------------------------------------------------------------------
src/__init__.py 0 0 100%
src/config/__init__.py 0 0 100%
src/config/settings.py 26 0 100%
src/core/__init__.py 0 0 100%
src/core/entities/__init__.py 0 0 100%
src/core/entities/admin.py 5 0 100%
src/core/entities/assignment.py 24 3 88% 22, 25, 28
src/core/entities/audit_log.py 15 0 100%
src/core/entities/course.py 25 0 100%
src/core/entities/draft.py 17 0 100%
src/core/entities/embedding.py 12 0 100%
src/core/entities/enrollment.py 25 0 100%
src/core/entities/file.py 16 0 100%
src/core/entities/hint.py 18 0 100%
src/core/entities/instructor.py 52 12 77% 37-61, 66, 74, 76, 85
src/core/entities/notification.py 23 5 78% 5, 19-20, 22-23
src/core/entities/peer_review.py 24 1 96% 34
src/core/entities/remediation.py 49 0 100%
src/core/entities/result.py 25 0 100%
src/core/entities/sandbox_job.py 43 0 100%
src/core/entities/similarity_comparison.py 13 0 100%
src/core/entities/similarity_flag.py 30 12 60% 17-20, 23-26, 28-31
src/core/entities/student.py 10 0 100%
src/core/entities/submission.py 29 0 100%
src/core/entities/test_case.py 28 1 96% 29
src/core/entities/user.py 36 2 94% 40, 42
src/core/exceptions/__init__.py 0 0 100%
src/core/exceptions/auth_error.py 10 1 90% 25
src/core/exceptions/validation_error.py 13 2 85% 12, 16
src/core/services/__init__.py 0 0 100%
src/core/services/admin_service.py 57 5 91% 70, 73, 85, 108-109
src/core/services/assignment_service.py 68 0 100%
src/core/services/audit_log_service.py 16 0 100%
src/core/services/auth_service.py 48 0 100%
```

src/core/services/course_service.py 20 0 100%
src/core/services/draft_service.py 19 0 100%
src/core/services/embedding_service.py 39 0 100%
src/core/services/enrollment_service.py 38 0 100%
src/core/services/file_service.py 41 0 100%
src/core/services/hint_service.py 28 0 100%
src/core/services/instructor_service.py 76 0 100%
src/core/services/notification_service.py 38 0 100%
src/core/services/peer_review_service.py 79 0 100%
src/core/services/remediation_service.py 87 0 100%
src/core/services/result_service.py 25 0 100%
src/core/services/sandbox_service.py 121 0 100%
src/core/services/similarity_flag_service.py 83 0 100%
src/core/services/similarity_service.py 88 0 100%
src/core/services/student_service.py 88 0 100%
src/core/services/submission_service.py 46 0 100%
src/core/services/test_case_service.py 57 0 100%
src/infrastructure/__init__.py 0 0 100%
src/infrastructure/ai/__init__.py 0 0 100%
src/infrastructure/ai/gemini_client.py 20 0 100%
src/infrastructure/ai/groq_client.py 44 11 75% 11-13, 57, 68-83, 86
src/infrastructure/database/__init__.py 0 0 100%
src/infrastructure/database/connection.py 61 0 100%
src/infrastructure/database/create_db.py 20 1 95% 12
src/infrastructure/database/migrations.py 37 4 89% 14, 42-43, 55
src/infrastructure/repositories/__init__.py 0 0 100%
src/infrastructure/repositories/admin_repository.py 35 0 100%
src/infrastructure/repositories/assignment_repository.py 73 0 100%
src/infrastructure/repositories/audit_log_repository.py 40 0 100%
src/infrastructure/repositories/course_repository.py 66 4 94% 171-179
src/infrastructure/repositories/database.py 59 0 100%
src/infrastructure/repositories/draft_repository.py 34 0 100%
src/infrastructure/repositories/embedding_repository.py 30 0 100%
src/infrastructure/repositories/enrollment_repository.py 53 4 92% 134-142
src/infrastructure/repositories/file_repository.py 38 0 100%
src/infrastructure/repositories/hint_repository.py 60 0 100%
src/infrastructure/repositories/instructor_repository.py 40 0 100%
src/infrastructure/repositories/notification_repository.py 39 0 100%
src/infrastructure/repositories/peer_review_repository.py 58 0 100%
src/infrastructure/repositories/remediation_repository.py 60 0 100%
src/infrastructure/repositories/result_repository.py 30 0 100%
src/infrastructure/repositories/sandbox_job_repository.py 36 0 100%
src/infrastructure/repositories/similarity_comparison_repository.py 49 0 100%
src/infrastructure/repositories/similarity_flag_repository.py 82 0 100%
src/infrastructure/repositories/student_repository.py 42 0 100%
src/infrastructure/repositories/submission_repository.py 80 0 100%
src/infrastructure/repositories/test_case_repository.py 44 0 100%
src/infrastructure/repositories/user_repository.py 74 0 100%

src/web/__init__.py 0 0 100%
src/web/app.py 180 13 93% 84, 96, 106, 116, 128-129, 319, 322-324, 329, 338-339
src/web/routes/__init__.py 0 0 100%
src/web/routes/admin.py 62 0 100%
src/web/routes/api.py 115 0 100%
src/web/routes/assignment.py 84 0 100%
src/web/routes/audit_log.py 54 0 100%
src/web/routes/auth.py 57 0 100%
src/web/routes/course.py 162 4 98% 79-80, 115-116
src/web/routes/draft.py 0 0 100%
src/web/routes/enrollment.py 103 0 100%
src/web/routes/file.py 73 0 100%
src/web/routes/hint.py 46 0 100%
src/web/routes/instructor.py 189 0 100%
src/web/routes/notification.py 47 0 100%
src/web/routes/peer_review.py 50 0 100%
src/web/routes/remediation.py 77 0 100%
src/web/routes/result.py 16 0 100%
src/web/routes/student.py 168 0 100%
src/web/routes/submission.py 34 0 100%
src/web/routes/test_case.py 50 0 100%
src/web/utils.py 48 10 79% 24-25, 40-41, 44-45, 56-57, 60-61

TOTAL 4649 95 98%
Coverage HTML written to dir htmlcov
Coverage XML written to file coverage.xml
========== **952 passed** in 30.24s ==========