UNIVERSITÀ DEGLI STUDI DI SALERNO

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE ED ELETTRICA E MATEMATICA APPLICATA

Report - Agile Software Processes and Devops

Peer Review System: PeerFlow
Software Development Report
Group 1

Professors

Nicola Capuano - ncapuano@unisa.it Francesco Moscato - fmoscato@unisa.it

Team members

Name and Surname	Student ID	E-mail
Ciaravola Giosuè	0622702177	g.ciaravola3@studenti.unisa.it
Della Corte Mario	0622702354	m.dellacorte19@studenti.unisa.it
Polverino Alessandro	0622702352	a.polverino15@studenti.unisa.it

ACADEMIC YEAR 2024/2025

Contents

1 CI/CD Pipeline

The CI/CD pipeline operates across a **Local Environment** (Windows machine) and a **Remote Environment** (Ubuntu VM). This setup utilizes a comprehensive stack of tools to efficiently manage the entire system life-cycle.

1.1 Local Environment

The **Local Environment** on a Windows machine serves as the initial stage for development and testing. It comprises:

- **FastAPI + Vue.js**: The core application, consisting of a FastAPI backend and an Vue.js frontend.
- **Postman**: Used to define integration and load tests.
- **Local Git repository**: For version control of the project code.
- **Local Git hooks**: Configured to automate tasks during local development:
  - **Pre-commit hook**:
    - Runs **unit tests**: in a Docker container, the code is loaded and pytest runs the unit tests and reports the results.
  - **Pre-push hook**:
    - Aborts any push to the `production` branch. This ensures that only Jenkins on the remote environment can push automatically to `production`. This is important for the following reasons:
      - this enforces the GitFlow branching strategy, in which developer changes must be added to the repository as features of the `dev` branch, in order to be properly tested; the developer cannot introduce changes to the production environment directly;
      - this ensures that the fast-forward merge between the `release` and the `production` branches will always proceed without conflicts.

1.2 Remote Environment

The **Remote Environment** consists of a ”on premise” VM accessible via VPN and SSH connection, responsible for automated testing and deployment. It includes:

- **Remote Git repository**: The central hub for all code changes.
- **Remote Git hooks**:
  - **Post-receive hook**:
    - **Triggers the Jenkins pipeline** upon a successful push.
- **Jenkins**: The automation server managing the CI/CD workflow based on branch activity.
- **Kubernetes - MicroK8s**: A simple single node Kubernetes cluster, which will host both the test and the production environments.
- **NodeJS - Newman**: CLI tool to run Postman collections automatically in the CI/CD Pipeline.

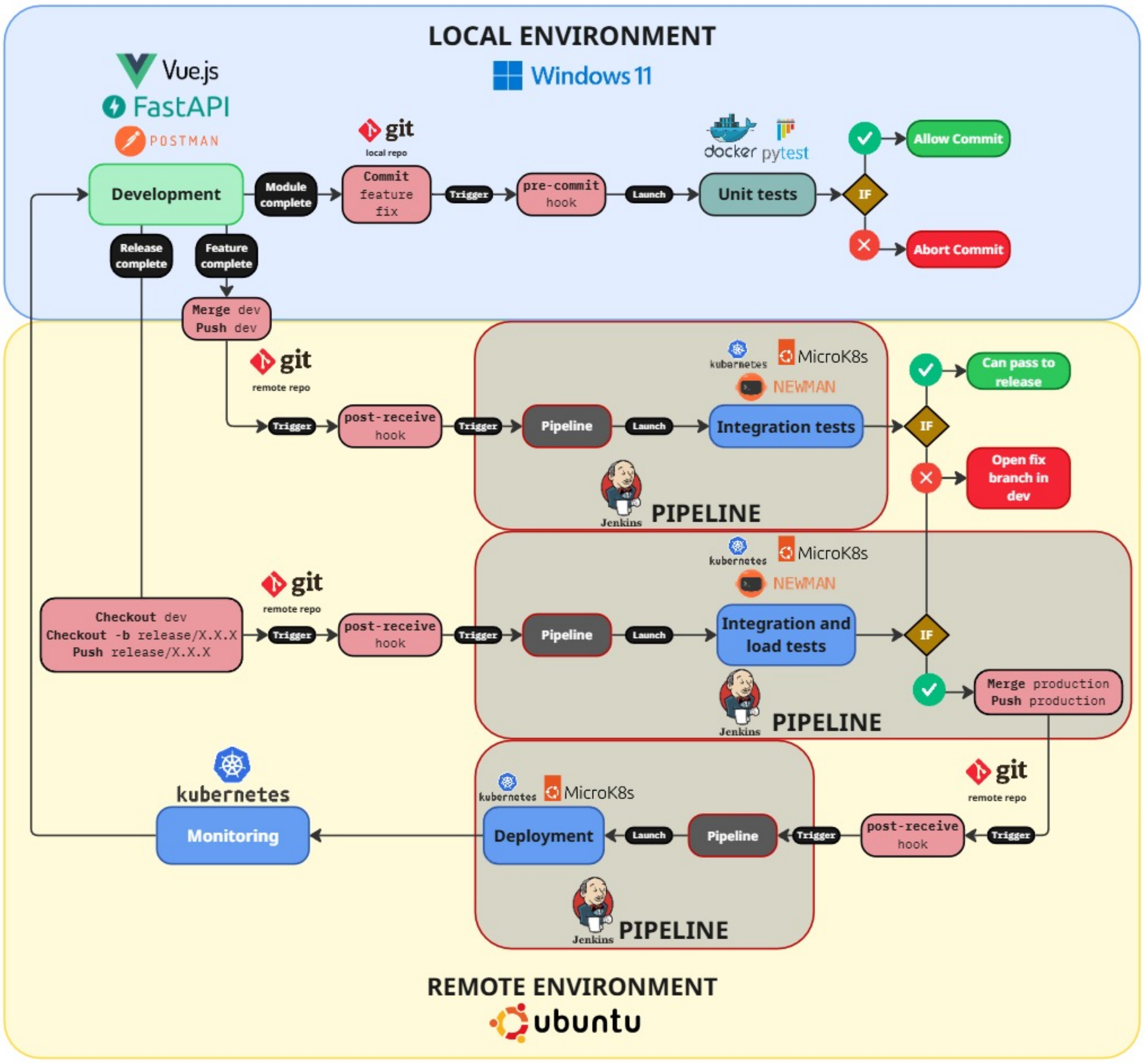

Figure 1.1: CI/CD Complete Pipeline.

1.3 Complete developer workflow

1.3.1 Local Development and Unit Tests

- Developers work on `feature` branches derived from `dev`.
- A **local pre-commit hook** runs all unit tests using `pytest` within a Docker container before any changes are committed.
- The commit is only allowed if all unit tests pass successfully.
- A commit message template is used to ensure that commit messages are clear, consistent, and informative.

1.3.2 Local Integration Tests

- **Integration tests** are executed locally **before pushing** to the remote repository. This allows developers to catch integration issues early and ensures that the integration tests are well written.

1.3.3 Completion of a feature and integration tests

- A completed feature branch is **merged into `dev`**.
- The `dev` branch is then **pushed to the remote repository**, triggering automatic integration tests.
  - If they fail, the developer can create a `fix` branch from `dev` and then merge and push to `dev` (more suitable for fixes that require more commits), or he can fix locally and push directly to `dev` (more suitable for a quick fix).

1.3.4 Completion of a release, load tests and automatic deployment

- Once a group of features have been tested and are considered ready to be released in production, a developer **creates and checks out to a `release/X.X.X` branch**.
- This `release/X.X.X` branch is then **pushed to the remote repository**, triggering automatic integration and load tests.
  - If they succeed, the `release/X.X.X` branch is automatically merged to `production` and the system is automatically deployed.
  - If they fail, the developer can *cry in a corner because the VM sucks* open a fix branch.

1.3.5 Post-deployment and monitoring

- The developers can monitor the state of the Kubernetes cluster using the included **dashboard**, which also shows resource usage and performance metrics for each pod.
- The developers can always visualize a complete report of each series of integration and load tests, which is automatically generated by Newman, using the Jenkins interface. This is particularly useful to troubleshoot in case of failure.

1.4 Jenkins automation script

The automation of the pipeline is made possible by a `Jenkinsfile` included in the Git repository, which executes specific actions based on the branch that triggered the script.

- **If branch is `fix` or `feature`**:
  - Jenkins takes **no action**.
- **If branch is `dev`**:
  - The Docker images of the microservices are built with tag *test* and pushed to DockerHub.
  - Kubernetes uses the images from DockerHub and the deployment specifications to deploy the application in the test environment (it has its specific namespace, so the test and production resources are separated).
  - **Integration tests** are run using **Newman** (for Postman collections).
  - If tests fail the pipeline fails alerting the developers.
  - If tests pass the pipeline completes successfully.
  - In any case the deployment is eliminated to release the resources.
- **If branch is `release`**:
  - The Docker images of the microservices are built with tag *test* and pushed to DockerHub.
  - Kubernetes uses the images from DockerHub and the deployment specifications to deploy the application in the test environment (it has its specific namespace, so the test and production resources are separated).
  - **Integration and load tests** are run using **Newman**.
  - If tests fail the pipeline fails alerting the developers.
  - If tests pass:
    - Jenkins performs a **fast-forward merge from `release` to `production`**. This simulates a pre-receive hook's function as Jenkins acts as the authorized pusher.
    - Jenkins then **pushes the changes to the `production` branch**, triggering another instance of the pipeline for which the branch is `production`.
  - In any case the deployment is eliminated to release the resources.
- **If branch is `production`**:
  - The Docker images of the microservices are built with tag *prod* and pushed to DockerHub.
  - Kubernetes uses the images from DockerHub and the deployment specifications to deploy the application in the production environment (it has its specific namespace, so the test and production resources are separated).
    - Since the production deployment is always up, a new deployment triggers a rolling upgrade that minimizes downtime.

2 DevOps Pratices

This chapter details the main DevOps practices. For each practice, a definition is provided, followed by an explanation of how it is enacted within the described CI/CD pipeline.

- **Build Automation**
  - *Definition:* Build Automation is the practice of automating the process of preparing code for deployment to a live environment. The specific tools used are often tied to the programming language or platform selected. Its benefits include creating fast, consistent, and repeatable builds that are more reliable than manual processes.
  - *Enactment in the Pipeline:* The pipeline automates code preparation at multiple stages. Locally, pre-commit hooks initiate automated unit testing within Docker. In the remote environment, Jenkins automates the creation of version-tagged Docker images for each microservice upon code pushes to relevant branches, making these artifacts ready for deployment.
- **Continuous Integration (CI)**
  - *Definition:* Continuous Integration (CI) is a DevOps practice involving frequent code merging by developers into a central repository, followed by automated builds and tests. Key benefits include early bug detection (like compilation errors), maintaining code in a deployable state, and encouraging modular code.
  - *Enactment in the Pipeline:* The pipeline embodies CI through a workflow where developers frequently merge feature branches into a central `dev` branch. This triggers a sequence of automated tests: unit tests run locally before commits, and upon pushing to `dev`, Jenkins orchestrates integration tests in a dedicated Kubernetes test environment, ensuring early feedback on code stability.
- **Continuous Deployment (CD)**
  - *Definition:* Continuous Deployment (CD) is the practice of automatically deploying small code changes to production in a routine and frequent manner once they have passed all automated tests. Benefits include faster time to market, a dependable deployment process, and reliable rollbacks.
  - *Enactment in the Pipeline:* The pipeline achieves CD by automating the path to production. Successful integration and load testing on a `release` branch trigger a Jenkins workflow that automatically merges code into the `production` branch. This, in turn, initiates an automated deployment of the updated application to the live Kubernetes environment, using rolling upgrades to ensure service continuity.

- **Infrastructure as Code (IaC)**
  - *Definition:* Infrastructure as Code (IaC) is the practice of managing and provisioning IT infrastructure through code and automation, rather than manual processes. This leads to consistent resource creation, reusability, self-documenting infrastructure, and simplification of complex setups.
  - *Enactment in the Pipeline:* IaC is central to the pipeline, with Dockerfiles defining application runtime environments and Kubernetes manifests specifying the deployment infrastructure (namespaces, services, etc.). All these are version-controlled in Git. Jenkins uses these codified definitions to automate the consistent provisioning and configuration of both test and production environments within MicroK8s.
- **Configuration Management**
  - *Definition:* Configuration Management involves managing and changing the state of infrastructure in constant and maintainable ways. Benefits include saving time, providing insight into infrastructure, maintainability with system changes, and minimizing configuration drift, especially in large environments.
  - *Enactment in the Pipeline:* The pipeline manages configurations by versioning all critical definitions—Dockerfiles, Kubernetes manifests, and the Jenkinsfile—in Git. Jenkins applies these configurations consistently when creating environments. Strict controls, such as restricted direct pushes to the `production` branch and the use of distinct Kubernetes namespaces, further ensure configuration integrity and prevent drift.
- **Orchestration**
  - *Definition:* Orchestration refers to the automation that supports processes and workflows, such as resource provisioning, often coordinating multiple automated tasks. It brings benefits like scalability, stability (especially with automatic responses to problem detection), and time savings.
  - *Enactment in the Pipeline:* The entire CI/CD process is orchestrated. Jenkins, guided by the `Jenkinsfile`, manages the end-to-end workflow including builds, multi-stage testing, and deployments across different branches. Kubernetes orchestrates the runtime lifecycle of the containerized application services. Git hooks contribute by automating the initial triggers for local tests and remote pipeline execution.
- **Monitoring**
  - *Definition:* Monitoring involves collecting and presenting data about the performance and stability of services and infrastructure, as well as detecting problems. Benefits include fast recovery, more data for root-cause analysis, cross-team visibility, and enabling automated responses.
  - *Enactment in the Pipeline:* The pipeline provides operational insight through several mechanisms. Jenkins makes detailed reports from automated integration and load tests (via Newman) available for analysis. Developers can observe real-time application performance and resource utilization using the Kubernetes dashboard. The Jenkins system itself offers status updates and alerts on the health of the pipeline execution.

3 Test Cases

3.1 Unit Tests

3.1.1 Authentication & Profiling Service

Table 3.1: Authentication & Profiling Service Test Cases and Descriptions

Test Case ID	Description
UT-SYS-001	Verifies the root endpoint of the application, ensuring it returns a successful response and the expected welcome message.
UT-SYS-002	Verifies the health check endpoint, ensuring it indicates a healthy status and a smooth operation message.
UT-SYS-003	Verifies the public key endpoint, ensuring that the endpoint returns a public key string upon successful generation or an error if key generation fails.
UT-SYS-004	Verifies successful user signup. It ensures that a new user can be registered with valid credentials and that the response contains the user's details.
UT-SYS-005	Verifies signup with a duplicate email. It ensures that the system prevents the creation of multiple accounts with the same email address.
UT-SYS-006	Verifies signup with invalid data. It ensures that the system handles missing or improperly formatted signup information.
UT-SYS-007	Verifies successful user login. It ensures that a registered user can log in with correct credentials and receive access and refresh tokens.
UT-SYS-008	Verifies login with an invalid email. It ensures that the system rejects login attempts with non-existent email addresses.
UT-SYS-009	Verifies login with an invalid password. It ensures that the system rejects login attempts with incorrect passwords for existing users.

Continued on next page

Table 3.1 – continued from previous page

Test Case ID	Description
UT-SYS-010	Verifies successful token refresh. It ensures that a valid refresh token can be used to obtain new access and refresh tokens.
UT-SYS-011	Verifies token refresh with an access token. It ensures that the system correctly identifies and rejects attempts to refresh using an access token instead of a refresh token.
UT-SYS-012	Verifies requests to non-existent endpoints. It ensures that the application returns a 404 status for unknown routes.
UT-SYS-013	Verifies invalid HTTP methods on existing endpoints. It ensures that the application correctly handles unsupported HTTP methods for specific routes.
UT-SYS-014	Verifies endpoints with malformed JSON. It ensures that the application properly handles requests with invalid JSON payloads.
UT-SYS-015	Verifies endpoints with missing content type. It ensures that the application gracefully handles requests where the content type header is absent.
UT-SYS-016	Verifies the presence of CORS headers. It ensures that the application includes appropriate Cross-Origin Resource Sharing headers in its responses.
UT-SYS-017	Verifies the creation and verification of JWT tokens. It ensures that access tokens can be successfully generated and their payloads correctly verified.
UT-SYS-018	Verifies successful retrieval of the current user's profile. It ensures that an authenticated user can fetch their own details.
UT-SYS-019	Verifies retrieval of the current user without a token. It ensures that access to the current user endpoint is denied without proper authentication.
UT-SYS-020	Verifies successful retrieval of students. It ensures that the system can list users with the 'Student' role.
UT-SYS-021	Verifies retrieval of students when no students exist. It ensures that the system returns an empty list if no users with the 'Student' role are found.
UT-SYS-022	Verifies successful retrieval of a user by ID. It ensures that a specific user's profile can be fetched using their unique identifier.
UT-SYS-023	Verifies retrieval of a user by ID when the user doesn't exist. It ensures that the system correctly indicates when a requested user ID is not found.

Continued on next page

Table 3.1 – continued from previous page

Test Case ID	Description
UT-SYS-024	Verifies successful batch retrieval of users. It ensures that multiple user profiles can be fetched simultaneously using a list of IDs.
UT-SYS-025	Verifies batch retrieval with some valid and some invalid IDs. It ensures that the system returns only the valid user profiles when a mix of IDs is provided.
UT-SYS-026	Verifies batch retrieval without a request body. It ensures that the system handles requests to the batch user retrieval endpoint that lack the necessary payload.

3.1.2 Assignment Service

Table 3.2: Assignment Service Test Cases and Descriptions

Test Case ID	Description
UT-ASG-001	This test case verifies that the root endpoint is accessible. It ensures the service returns a welcome message.
UT-ASG-002	This test case verifies that the health check endpoint is accessible. It ensures the service reports a healthy status.
UT-ASG-003	This test case verifies that retrieving all assignments returns an empty list when no assignments exist. It ensures the service handles the absence of records gracefully.
UT-ASG-004	This test case verifies that a specific assignment can be successfully retrieved by its ID. It ensures the service provides the correct assignment details.
UT-ASG-005	This test case verifies that retrieving a non-existent assignment by ID results in an appropriate error. It ensures the service correctly handles requests for unknown assignments.
UT-ASG-006	This test case verifies that assignments can be successfully retrieved for a specific teacher. It ensures the service provides a list of assignments associated with the teacher.
UT-ASG-007	This test case verifies that retrieving assignments for a teacher with no assigned tasks returns an empty list. It ensures the service correctly handles scenarios where a teacher has no assignments.

Continued on next page

Table 3.2 – continued from previous page

Test Case ID	**Description**
UT-ASG-008	This test case verifies that assignments can be successfully retrieved for a specific student. It ensures the service provides a list of assignments associated with the student.
UT-ASG-009	This test case verifies that retrieving assignments for a student with no assigned tasks returns an empty list. It ensures the service correctly handles scenarios where a student has no assignments.
UT-ASG-010	This test case verifies that assignment creation fails gracefully when a database error occurs. It ensures the service handles database insertion failures.
UT-ASG-011	This test case verifies that assignment creation fails when provided with invalid data. It ensures the service enforces data validation rules.
UT-ASG-012	This test case verifies that an existing assignment can be successfully updated. It ensures the service correctly modifies assignment details.
UT-ASG-013	This test case verifies that attempting to update a non-existent assignment results in an appropriate error. It ensures the service correctly handles updates for unknown assignments.
UT-ASG-014	This test case verifies that assignment updates fail gracefully when a database error occurs. It ensures the service handles database update failures.
UT-ASG-015	This test case verifies that multiple assignments can be retrieved for a teacher. It ensures the service correctly returns all relevant assignments.
UT-ASG-016	This test case verifies that multiple assignments can be retrieved for a student. It ensures the service correctly returns all relevant assignments.
UT-ASG-017	This test case verifies that an assignment can be partially updated. It ensures the service correctly applies partial modifications.

3.1.3 Assignment Submission Service

Table 3.3: Assignment Submission Service Test Cases and Descriptions

Test Case ID	**Description**
UT-ASG-SUB-001	This test case verifies that the root endpoint returns the correct service message. Tests basic API connectivity.
UT-ASG-SUB-002	This test case verifies that the health check endpoint returns healthy status. Tests service health monitoring functionality.

Continued on next page

Table 3.3 – continued from previous page

Test Case ID	**Description**
UT-ASG-SUB-003	This test case verifies successful retrieval of submission data by submission ID. Tests database query functionality for individual submissions.
UT-ASG-SUB-004	This test case verifies successful retrieval of multiple submissions by assignment ID. Tests database query functionality for assignment-related submissions.
UT-ASG-SUB-005	This test case verifies proper handling when no submissions exist for an assignment. Tests empty result scenarios in submission queries.
UT-ASG-SUB-006	This test case verifies successful upload and creation of new assignment submissions. Tests submission creation functionality with valid data.
UT-ASG-SUB-007	This test case verifies proper handling when attempting to upload duplicate submissions. Tests duplicate prevention logic in submission creation.
UT-ASG-SUB-008	This test case verifies proper error handling when database insertion fails. Tests database error scenarios during submission creation.
UT-ASG-SUB-009	This test case verifies successful retrieval of specific student submissions by assignment and student ID. Tests targeted submission queries.
UT-ASG-SUB-010	This test case verifies proper handling when student hasn't submitted for an assignment. Tests missing submission scenarios.
UT-ASG-SUB-011	This test case verifies proper validation when required submission data is missing. Tests input validation for incomplete data.
UT-ASG-SUB-012	This test case verifies proper validation when invalid file types are provided. Tests file type validation in submission attachments.
UT-ASG-SUB-013	This test case verifies successful submission upload with empty attachments list. Tests submission creation with minimal attachment data.
UT-ASG-SUB-014	This test case verifies proper validation when required query parameters are missing. Tests parameter validation for submission retrieval.
UT-ASG-SUB-015	This test case verifies assignment submission upload functionality through integration testing. Tests end-to-end submission creation process.

3.1.4 Review Assignment Service

Table 3.4: Peer Review Assignment Service Test Cases and Descriptions

Test Case ID	**Description**
UT-PRW-001	This test case verifies that the root endpoint returns the correct welcome message. This test case verifies the basic functionality of the application's main route.
UT-PRW-002	This test case verifies that the health check endpoint responds appropriately. This test case verifies the application's health monitoring capability.
UT-PRW-003	This test case verifies that all peer review assignments can be retrieved successfully. This test case verifies the functionality to fetch multiple peer review records.
UT-PRW-004	This test case verifies that retrieving all peer reviews returns an empty list when no assignments exist. This test case verifies the behavior when the database contains no peer review data.
UT-PRW-005	This test case verifies that peer reviews can be retrieved by providing a batch of assignment IDs. This test case verifies the batch retrieval functionality for multiple specific assignments.
UT-PRW-006	This test case verifies that batch retrieval returns an empty list when no matching assignments are found. This test case verifies the behavior when batch queries yield no results.
UT-PRW-007	This test case verifies that a specific peer review assignment can be retrieved by assignment ID. This test case verifies the single assignment retrieval functionality.
UT-PRW-008	This test case verifies that retrieving a non-existent peer review assignment is handled properly. This test case verifies the error handling for missing assignment data.
UT-PRW-009	This test case verifies that retrieving a peer review assignment handles missing rubric data appropriately. This test case verifies the error handling when associated rubric information is unavailable.
UT-PRW-010	This test case verifies that a new peer review assignment can be created successfully. This test case verifies the complete creation workflow including rubric and assignment data.
UT-PRW-011	This test case verifies that creating a peer review assignment prevents duplicate assignments. This test case verifies the validation logic for existing assignment IDs.
UT-PRW-012	This test case verifies that peer review assignment creation handles rubric creation failures. This test case verifies the error handling during the rubric creation process.

Continued on next page

Table 3.4 – continued from previous page

Test Case ID	**Description**
UT-PRW-013	This test case verifies that peer review assignment creation handles assignment creation failures. This test case verifies the error handling during the peer review assignment creation process.
UT-PRW-014	This test case verifies that peer review results can be submitted successfully. This test case verifies the functionality to record review outcomes and scores.
UT-PRW-015	This test case verifies that submitting peer review results handles cases with no existing pairings. This test case verifies the error handling when no review pairings are configured.
UT-PRW-016	This test case verifies that submitting peer review results handles cases where the specified pairing doesn't exist. This test case verifies the validation of reviewer-reviewee relationships.
UT-PRW-017	This test case verifies that peer review assignments can be updated successfully. This test case verifies the modification functionality for existing assignments.
UT-PRW-018	This test case verifies that peer review assignment updates handle database operation failures. This test case verifies the error handling during the update process.
UT-PRW-019	This test case verifies that a new peer review assignment can be created through the assignment endpoint. This test case verifies the alternative creation pathway for peer review assignments.

3.1.5 Review Processing Service

Table 3.5: Results & Monitoring Service Test Cases and Descriptions

Test Case ID	**Description**
UT-RES-001	This test case verifies that the root endpoint returns the correct welcome message. It ensures the main application entry point is functioning properly.
UT-RES-002	This test case verifies that the health check endpoint returns the proper health status. It confirms the service monitoring functionality is operational.
UT-RES-003	This test case verifies that the statistics calculation endpoint processes valid review pairings successfully. It ensures the core statistical processing functionality works with complete review data.

Continued on next page

Table 3.5 – continued from previous page

Test Case ID	**Description**
UT-RES-004	This test case verifies that the statistics calculation endpoint handles empty pairings lists correctly. It ensures the system gracefully processes scenarios with no review data.
UT-RES-005	This test case verifies that the statistics calculation endpoint processes in-progress reviews appropriately. It ensures the system correctly handles incomplete review submissions.
UT-RES-006	This test case verifies that the aggregated assignment data retrieval endpoint returns correct information for existing assignments. It ensures assignment-level statistics can be successfully retrieved.
UT-RES-007	This test case verifies that the aggregated assignment data retrieval endpoint handles non-existent assignments properly. It ensures appropriate error handling for missing assignment data.
UT-RES-008	This test case verifies that the aggregated submission data retrieval endpoint returns correct information for existing submissions. It ensures submission-level statistics can be successfully retrieved.
UT-RES-009	This test case verifies that the aggregated submission data retrieval endpoint handles non-existent submissions properly. It ensures appropriate error handling for missing submission data.
UT-RES-010	This test case verifies that the aggregated review data retrieval endpoint returns all reviews for a specific submission. It ensures comprehensive review data retrieval functionality.
UT-RES-011	This test case verifies that the aggregated review data retrieval endpoint returns specific reviewer-submission pair data. It ensures targeted review data retrieval functionality.
UT-RES-012	This test case verifies that the aggregated review data retrieval endpoint handles submissions with no reviews properly. It ensures appropriate error handling when no review data exists.
UT-RES-013	This test case verifies that the aggregated review data retrieval endpoint handles non-existent reviewer-submission pairs properly. It ensures appropriate error handling for missing specific review combinations.
UT-RES-014	This test case verifies that the statistics calculation endpoint handles malformed JSON input appropriately. It ensures proper validation of request data format.
UT-RES-015	This test case verifies that the statistics calculation endpoint handles missing required parameters correctly. It ensures proper validation of required input parameters.

3.2 Integration Tests

Table 3.6: Integration Test Cases and Descriptions

Test Case ID	**Description**
UT-INT-001	This test case verifies that students can retrieve their assigned assignments. The test ensures proper assignment listing functionality for student users.
UT-INT-002	This test case verifies that teachers can create new assignments with involved students. The test validates assignment creation functionality with dynamic deadline setting.
UT-INT-003	This test case verifies that students can view detailed information about specific assignments. The test ensures proper assignment detail retrieval for student users.
UT-INT-004	This test case verifies that teachers can view comprehensive assignment details including submissions and peer reviews. The test validates enhanced assignment information access for teachers.
UT-INT-005	This test case verifies that students can submit their assignment work with text content. The test ensures proper assignment submission functionality.
UT-INT-006	This test case verifies that students can retrieve their own submission for a specific assignment. The test validates submission retrieval functionality for student users.
UT-INT-007	This test case verifies that teachers can view all submissions for a specific assignment. The test ensures comprehensive submission overview for teachers.
UT-INT-008	This test case verifies that students can access peer review assignments when available. The test validates peer review functionality and handles cases where no peer review exists.
UT-INT-009	This test case verifies that teachers can close assignment submissions by updating the deadline. The test ensures proper assignment status management and deadline enforcement.
UT-INT-010	This test case verifies that teachers can create peer review assignments with rubrics and deadlines. The test validates peer review setup functionality for collaborative assessment.

3.3 Load tests

This load test validates the PeerFlow educational platform's performance and scalability under concurrent user load by simulating 100 students registering, authenticating, and submitting assignments simultaneously while ensuring sub-500ms response times across all operations.

- **Test Workflow**
  - **Phase 1: Setup**
    - *Teacher Account Creation:* Create teacher account with unique credentials.
    - *Teacher Authentication:* Log in as the teacher and obtain an authentication token.
  - **Phase 2: Student Registration Wave**
    - *Student Registration (100 iterations):*
      - Register unique student accounts with generated credentials.
      - **Delay:** 1ms between each registration.
      - Store student IDs and credentials for subsequent phases.
  - **Phase 3: Authentication Storm**
    - *Student Login (100 iterations):*
      - Authenticate all registered students.
      - **Delay:** 1ms between each login.
      - Capture and store authentication tokens.
  - **Phase 4: Assignment Creation**
    - *Assignment Creation:* Teacher creates an assignment for all 100 students.
      - Set a 7-day submission deadline.
      - Include all registered student IDs.
  - **Phase 5: Submission Flood**
    - *Assignment Submission (100 iterations):*
      - All students submit unique content to the assignment.
      - **Delay:** 1ms between each submission.
      - Generate timestamped submission content.
- **Validation Checks**
  - **Performance Validation**
    - *Response Time:* All API calls must complete within 750ms.
    - *Status Codes:* Verify successful HTTP status codes (200, 201) for all operations.

List of Figures

List of Tables