TED UNIVERSITY

CMPE 492/SENG 492 Senior Project II

SyntaxSavior Test Plan Report

13.05.2025

Team Members:

İrem Beşiroğlu   Sofware Engineering

Yiğit Oğuzhan Kökten Computer Engineering

Yüksel Çağlar Baypınar Computer Engineering

Table of Contents

1. Introduction

This Test Plan Report outlines the scope, objectives, and methodology for testing the SyntaxSavior system, an educational platform designed to assist students in introductory programming courses (e.g., CMPE113) at TED University. The system integrates a Visual Studio Code (VS Code) plugin, a backend processing server, an AI-powered feedback mechanism, and a vector database to provide real-time code analysis, contextual feedback, and secure access control. The test plan ensures that the system meets functional, non-functional, and educational requirements while identifying and mitigating risks to deliver a reliable, scalable, and user-centric tool.

1.1. Scope

The testing scope encompasses all critical components and functionalities of the SyntaxSavior system, ensuring comprehensive validation of its features and interactions. The following areas are included in the testing process:

- **IDE Plugin Features**: Testing the VS Code plugin's user interface elements (e.g., syntax highlighting, error pop-ups), communication with the backend, and integration with local language servers or linters for real-time code monitoring.
- **Backend Processing and AI Feedback**: Validating the backend's code analysis capabilities, AI-driven intermediate feedback generation, and error categorization (syntax, runtime, logical). Note that AI feedback is not direct but serves as an interpretive layer for code analysis.
- **Vector Database Interaction**: Ensuring accurate retrieval of contextual course materials and explanations from the vector database (e.g., Milvus or ChromaDB).
- **Security Controls**: Verifying authentication, authorization, rate-limiting, anti-cheating mechanisms, and protection against jailbreaking or privilege escalation attempts through input filtering and validation.

- **User Roles and Data Flow**: Testing access control and data flow for four user roles: student, instructor, assistant, and administrator.
- **Feedback Accuracy**: Conducting non-automated user acceptance testing to evaluate the relevance and educational clarity of feedback provided to students.

Testing excludes external systems (e.g., TED University's Learning Management System) unless directly integrated with SyntaxSavior, and focuses on the system's core functionalities as defined in the high-level and low-level design reports.

1.2. Objectives

The primary objectives of the test plan are to ensure that SyntaxSavior meets its functional, non-functional, and educational goals while maintaining reliability, security, and usability. Specific objectives include:

- **Verify Functional Requirements**: Confirm that all system components (IDE plugin, backend, AI feedback, vector database) operate as specified in the high-level design, including real-time code monitoring, accurate error detection, and contextual feedback generation.
- **Validate Non-Functional Requirements**: Ensure scalability, performance (e.g., handling up to 50 concurrent submissions), security (e.g., jailbreak protection, encrypted data transfers), and usability for students and instructors.
- **Assess Educational Effectiveness**: Validate that feedback is curriculum-aligned, promotes learning, and avoids direct solutions to foster independent problem-solving, as per the system's pedagogical goals.
- **Identify and Mitigate Risks**: Detect potential issues such as server overload, incorrect feedback, or security vulnerabilities (e.g., unauthorized access or jailbreaking attempts) and implement mitigation strategies.

- **Ensure System Maintainability**: Verify that the system adheres to engineering standards (e.g., IEEE 1016-2009, SOLID principles) for ease of maintenance and future scalability.
- **Test IDE-Specific Features**: Confirm seamless integration with VS Code, including plugin responsiveness, error pop-up functionality, and compatibility with local linters.
- **Validate Security Against Jailbreaking**: Test input filtering and validation mechanisms to prevent malicious inputs or privilege escalation, ensuring robust protection for an educational environment.

By achieving these objectives, the test plan ensures that SyntaxSavior is a robust, secure, and effective tool for supporting programming education.

2. Features to be Tested

This section describes the key features of the SyntaxSavior system that will be tested, focusing on their functionality, performance, and alignment with educational goals. Each feature is mapped to specific components from the high-level and low-level design reports.

2.1. IDE Plugin

Description: The IDE Plugin, implemented as a VS Code extension, serves as the primary interface for students to interact with SyntaxSavior. It provides real-time code monitoring, syntax checking, and feedback display within the VS Code environment. The plugin communicates with the backend for in-depth analysis and integrates with local language servers or linters for surface-level error detection.

Features to Test:

- **Syntax Highlighting (Test ID: IDE-001)**: Verify that the plugin correctly highlights Java code syntax in VS Code, adhering to standard Java conventions.
- **Live Error Checking (Test ID: IDE-002)**: Ensure the plugin detects surface-level errors (e.g., missing semicolons, unmatched brackets) in real-time and displays them in the editor.
- **Pop-Ups for Suggestions (Test ID: IDE-003)**: Confirm that the plugin generates contextual pop-up suggestions for detected errors, including hints without direct solutions (e.g., "Add a semicolon to terminate the statement").
- **Backend Communication (Test ID: IDE-004)**: Validate bidirectional communication with the backend via REST API, ensuring code submissions are sent and feedback is received accurately.
- **UI Integration (Test ID: IDE-005)**: Test the plugin's UI elements (e.g., error highlights, feedback panels) for responsiveness and usability within VS Code.
- **Hybrid Monitoring (Test ID: IDE-006)**: Verify the hybrid monitoring approach, where manual triggers initiate in-depth analysis and automatic checks provide surface-level feedback, balancing user control and system efficiency.
- **Rationale**: These tests ensure that the plugin provides a seamless and educationally effective experience, aligning with the system's goal of fostering independent learning while offering timely assistance.

2.2. Backend Processing

Description: The Backend Processing subsystem, implemented using a Spring Boot-based Java server, handles code analysis, error categorization, and feedback generation. It processes code submissions from the IDE plugin, integrates with the

AI model for interpretive analysis, and retrieves contextual materials from the vector database.

Features to Test:

- **Code Analysis (Test ID: BCK-001)**: Verify that the CodeAnalysisEngine accurately identifies syntax, runtime, and logical errors in submitted Java code using Abstract Syntax Tree (AST) parsing and rule-based checks.
- **Error Categorization (Test ID: BCK-002)**: Ensure errors are correctly classified into syntax (e.g., missing semicolon), runtime (e.g., null pointer exception), and logical (e.g., incorrect loop logic) categories, with appropriate feedback for each.
- **AI Intermediate Feedback Querying (Test ID: BCK-003)**: Confirm that the backend queries the AI model (via LanguageModelInterface) to generate interpretive feedback, ensuring it aligns with CMPE113 curriculum objectives.
- **Request Handling (Test ID: BCK-004)**: Validate that the RequestHandler correctly processes incoming REST API requests, routes them to the CodeAnalysisEngine, and returns formatted responses.
- **Session Management (Test ID: BCK-005)**: Test the Session Manager's ability to track user sessions and maintain state during code submissions and feedback retrieval.
- **Curriculum Database Access (Test ID: BCK-006)**: Ensure the backend retrieves relevant course materials from the curriculum database to contextualize feedback.

Rationale: These tests validate the backend's core functionality, ensuring accurate code analysis, reliable feedback generation, and seamless integration with other subsystems, which are critical for educational effectiveness.

2.3. AI Intermediate Feedback

Description: The AI Intermediate Feedback feature leverages a fine-tuned Deepseek model to provide educational feedback based on code analysis results. The model interprets code errors and generates curriculum-aligned hints and explanations, avoiding direct solutions to promote learning.

Features to Test:

- **Feedback Generation (Test ID: AI-001)**: Verify that the LanguageModelInterface generates accurate, curriculum-aligned feedback for Java code errors, focusing on educational guidance (e.g., explaining why a semicolon is needed rather than providing the fix).
- **Fine-Tuning Effectiveness (Test ID: AI-002)**: Confirm that the fine-tuned Deepseek model correctly interprets CMPE113-specific Java tasks and provides relevant feedback based on course materials.
- **Contextual Relevance (Test ID: AI-003)**: Ensure feedback is tailored to the lab task and student's progress, using data from the curriculum database.
- **Non-Directive Feedback (Test ID: AI-004)**: Validate that the AI avoids providing direct code solutions, adhering to the system's pedagogical philosophy.
- **Error Handling Integration (Test ID: AI-005)**: Test the AI's ability to process and respond to various error types (syntax, runtime, logical) with appropriate explanations.
- **Performance Efficiency (Test ID: AI-006)**: Measure the latency of AI feedback generation to ensure it meets performance requirements (e.g., response within 2 seconds for typical queries).

Rationale: These tests ensure that the AI model delivers pedagogically sound feedback that enhances student understanding, aligns with course objectives, and maintains system responsiveness.

2.4. Vector Database

Description : The Vector Database subsystem (Milvus or ChromaDB) stores and retrieves semantically indexed course materials including lecture notes, sample problems, and instructor-written solutions and explanations. It helps the AI module provide contextually appropriate feedback by correlating code analysis findings with associated instructional information via vector embeddings. This subsystem guarantees that feedback is based on the CMPE113 curriculum and relevant to the student's current learning aim.

Features to Test:

- **Semantic Retrieval Accuracy (TEST ID: VD-001):** Check that the system obtains the most contextually appropriate articles based on the AI model's query vectors, guaranteeing alignment with the issue type and course topic.
- **Embedding Consistency (TEST ID: VD-002):** Confirm that document embeddings are consistently created and saved using the same vectorization process.
- **Query Performance (TEST ID: VD-003):** Measure the response times for semantic search queries to ensure that retrieval completes within acceptable latency criteria.
- **Data Update Handling (TEST ID: VD-004):** Validate that new educational content may be incorporated and added to the vector database without disturbing existing queries or creating inconsistency.

- **Contextual Relevance Scoring (TEST ID: VD-005):** Ensure that the system gives relevance scores to findings that are consistent with manuel evaluation by course assistants.
- **Access Control for Documents (TEST ID: VD-006):** Confirm that vector database access follows role-based permissions. For example, course assistants can upload or tag tasks, but students can only query through the feedback interface.
- **Curriculum Synchronization (TEST ID: VD-007):** Check that the vector database content fits the most recent CMPE113 curriculum and lab standards, preventing obsolete or misaligned feedback from being retrieved.

2.5. User Roles

Description: The SyntaxSavior system implements role-based access control (RBAC) to manage permissions for four user types: students, instructors, assistants, and administrators. Each role has distinct functionalities, such as code submission (students), feedback review and grading (instructors/assistants), and system configuration (administrators). The RBAC mechanism ensures secure and appropriate access to system features, protecting sensitive data and maintaining system integrity.

Features to Test:

- **Role-Based Authentication (Test ID: UR-001)**: Verify that users can log in using their credentials and are assigned the correct role (student, instructor, assistant, or administrator) based on the authentication service.
- **Access Control Enforcement (Test ID: UR-002)**: Confirm that each role is restricted to its authorized functionalities (e.g., students cannot access grading tools, administrators cannot submit code).

Rationale: These tests ensure that the vector database supports the production of meaningful and context-aware feedback appropriate for learning. Reliable, fast, and accurate retrieval of relevant content allows the AI feedback engine to base its guidance on course materials, improving learning effectiveness and student understanding. Curriculum updates and compliance with strict access control further ensure content integrity and role-appropriate interactions.

- **Student Permissions (Test ID: UR-003)**: Test that students can submit code, view feedback, and access curriculum-aligned materials but are blocked from administrative or instructor-specific features.
- **Instructor/Assistant Permissions (Test ID: UR-004)**: Validate that instructors and assistants can review student submissions, provide manual feedback, and access grading tools, but cannot modify system configurations.
- **Administrator Permissions (Test ID: UR-005)**: Ensure administrators can manage user accounts, configure system settings (e.g., rate limits, feedback rules), and monitor system logs, but are restricted from submitting or grading code.
- **Role Switching (Test ID: UR-006)**: Verify that users with multiple roles (e.g., an assistant who is also a student) can switch roles seamlessly without compromising access control.

Rationale: These tests ensure that the RBAC system enforces strict access controls, preventing unauthorized actions and protecting the system's educational integrity. Testing role-specific permissions aligns with security and usability requirements, ensuring a tailored experience for each user type.

2.6. Submission Checks

Description: The Submission Checks feature validates code submissions before processing to ensure they meet predefined criteria, such as correct file structure, package naming, and adherence to assignment requirements. These checks complement local linters by catching errors that are not typically detected locally, such as incorrect package declarations or missing submission metadata, reducing backend processing errors and improving user experience.

Features to Test:

- **File Structure Validation (Test ID: SC-001)**: Confirm that the system checks for correct file structure (e.g., .java files in the appropriate directory) before accepting submissions.

- **Package Naming Compliance (Test ID: SC-002)**: Verify that submissions adhere to expected package naming conventions (e.g., cmpe113.lab1) as defined in the assignment specifications.

- **Metadata Verification (Test ID: SC-003)**: Ensure the system validates submission metadata, such as student ID, lab number, and submission timestamp, to prevent incomplete or invalid submissions.

- **Pre-Submission Error Feedback (Test ID: SC-004)**: Test that the IDE plugin displays clear, actionable error messages (e.g., "Incorrect package name: expected cmpe113.lab1") when submission checks fail.

- **Bypass Prevention (Test ID: SC-005)**: Validate that users cannot bypass submission checks by manipulating inputs or using malformed submissions, ensuring robust validation.

- **Integration with Backend (Test ID: SC-006)**: Confirm that submission checks are performed consistently between the IDE plugin and backend, with no discrepancies in validation logic.

Rationale: These tests ensure that submission checks enhance system reliability by catching errors early, reducing backend load, and providing students with immediate feedback to correct submission issues. This aligns with the system's goal of fostering a smooth and educational user experience.

2.7. Feedback Accuracy

Description: The Feedback Accuracy feature ensures that the AI-generated and backend-processed feedback is relevant, educationally clear, and aligned with the CMPE113 curriculum. Feedback must guide students toward understanding errors and improving their code without providing direct solutions, promoting independent learning and critical thinking.

Features to Test:

- **Relevance to Error Type (Test ID: FA-001)**: Verify that feedback accurately addresses the specific error type (syntax, runtime, or logical) with appropriate explanations (e.g., "A null pointer exception occurs when accessing an uninitialized object").
- **Curriculum Alignment (Test ID: FA-002)**: Confirm that feedback references CMPE113 course materials and lab objectives, ensuring relevance to the student's learning context.
- **Educational Clarity (Test ID: FA-003)**: Test that feedback is written in clear, concise language suitable for novice programmers, avoiding technical jargon unless explained.
- **Non-Directive Guidance (Test ID: FA-004)**: Ensure feedback provides hints or explanations (e.g., "Check the loop termination condition") rather than direct code fixes to encourage problem-solving.
- **Consistency Across Submissions (Test ID: FA-005)**: Validate that similar errors in different submissions receive consistent feedback, ensuring fairness and reliability.
- **User Acceptance Feedback (Test ID: FA-006)**: Collect qualitative feedback from CMPE113 students and instructors during user acceptance testing to assess perceived accuracy and helpfulness.

Rationale: These tests ensure that feedback meets the system's pedagogical goals by being accurate, relevant, and supportive of learning. User acceptance testing validates educational effectiveness, while automated tests confirm consistency and alignment with course objectives.

2.8. Error Handling

Description: The Error Handling feature ensures that the SyntaxSavior system robustly detects, reports, and recovers from various error types (syntax, runtime, logical) during code analysis and user interactions. Effective error handling prevents system crashes, provides clear user feedback, and maintains operational stability under unexpected conditions.

Features to Test:

- **Syntax Error Reporting (Test ID: EH-001)**: Verify that syntax errors (e.g., missing semicolon, incorrect variable declaration) are detected and reported with precise location details and explanations.
- **Runtime Error Detection (Test ID: EH-002)**: Confirm that runtime errors (e.g., null pointer exceptions, array index out of bounds) are identified during code analysis with actionable feedback.
- **Logical Error Identification (Test ID: EH-003)**: Test the system's ability to flag logical errors (e.g., incorrect algorithm output) using rule-based checks and AI analysis, providing hints for correction.
- **Error Recovery (Test ID: EH-004)**: Ensure the system gracefully handles errors without crashing, allowing users to continue working (e.g., plugin remains responsive after a failed submission).
- **User-Friendly Error Messages (Test ID: EH-005)**: Validate that error messages are clear, concise, and tailored to novice programmers, avoiding cryptic technical details.

- **Logging and Monitoring (Test ID: EH-006)**: Confirm that all errors are logged with sufficient detail (e.g., error type, timestamp, user ID) for administrators to diagnose and resolve issues.

Rationale: These tests ensure that error handling is robust and user-friendly, maintaining system stability and providing students with clear guidance to resolve issues. Comprehensive logging supports maintainability and troubleshooting, aligning with engineering standards.

2.9. Security

Description: The Security feature protects the SyntaxSavior system against unauthorized access, malicious inputs, and cheating attempts, ensuring a secure educational environment. Security mechanisms include authentication, rate-limiting, anti-cheating measures, and input filtering to prevent jailbreaking or privilege escalation.

Features to Test:

- **Authentication (Test ID: SEC-001)**: Verify that only authenticated users with valid credentials can access the system, using secure protocols (e.g., OAuth 2.0 or JWT).
- **Authorization (Test ID: SEC-002)**: Confirm that role-based access controls prevent users from accessing unauthorized features or data (e.g., students cannot view other students' submissions).
- **Rate-Limiting (Test ID: SEC-003)**: Test that the system enforces rate limits (e.g., maximum submissions per minute) to prevent denial-of-service attacks or abuse.
- **Anti-Cheating Mechanisms (Test ID: SEC-004)**: Validate that the system detects and flags suspicious activities, such as identical code submissions or attempts to bypass feedback restrictions.

- **Input Filtering and Validation (Test ID: SEC-005)**: Ensure that all user inputs (e.g., code, metadata) are sanitized to prevent injection attacks, jailbreaking, or privilege escalation.
- **Data Encryption (Test ID: SEC-006)**: Confirm that sensitive data (e.g., user credentials, code submissions) is encrypted during transmission (e.g., TLS) and storage.

Rationale: These tests ensure that the system is secure against internal and external threats, protecting user data and maintaining the integrity of the educational process. Robust security measures are critical for an academic tool deployed in a university setting.

3. Testing Methodology

3.1. Unit Testing:

Description: Unit testing focuses on validating individual components or modules of the SyntaxSavior system in isolation, as specified in the low-level design report. The goal is to ensure that each unit (e.g., functions, classes, or methods) performs as expected under controlled conditions.

Approach:

- **Scope**: Test units such as the CodeAnalysisEngine's AST parsing logic, the LanguageModelInterface's feedback generation, the RequestHandler's API routing, and the IDE plugin's syntax highlighting and error detection functions.
- **Tools**: JUnit 5 for backend Java components, Jest for JavaScript-based plugin components, and Mockito for mocking dependencies.

- **Test Cases**: Develop test cases for each unit based on functional requirements (e.g., parsing a Java file, generating a feedback string) and edge cases (e.g., malformed input, null values).
- **Execution**: Automated tests run during continuous integration (CI) pipelines using Jenkins or GitHub Actions to ensure early defect detection.
- **Metrics**: Achieve at least 90% code coverage for critical modules, measured using JaCoCo for Java and Istanbul for JavaScript.

Rationale: Unit testing ensures that individual components are reliable and meet design specifications, reducing the likelihood of defects propagating to higher testing levels. This aligns with the system's maintainability and quality goals.

3.2. Integration Testing:

Description: Integration testing validates the interactions between SyntaxSavior's subsystems, including the IDE plugin, backend server, AI model, and vector database. The goal is to ensure seamless communication and data flow across components.

Approach:

- **Scope**: Test key integration points, such as:
    - IDE plugin to backend (REST API communication for code submission and feedback retrieval).
    - Backend to AI model (querying the Deepseek model via LanguageModelInterface).
    - Backend to vector database (retrieving contextual materials from Milvus/ChromaDB).
    - Authentication service to role-based access control (RBAC) enforcement.

- **Tools**: Postman for API testing, Selenium for plugin-backend interaction testing, and custom scripts for database query validation.
- **Test Cases**: Include scenarios for successful interactions (e.g., submitting valid code and receiving feedback) and failure cases (e.g., network errors, invalid API tokens).
- **Execution**: Conduct automated and manual tests in a staging environment mimicking production conditions.
- **Metrics**: Verify 100% coverage of critical integration paths, with zero critical defects in API or data exchange.

Rationale: Integration testing ensures that subsystems work together as intended, preventing issues such as data mismatches or communication failures. This is critical for the system's end-to-end functionality and user experience.

3.3. System Testing:

Description: System testing validates the SyntaxSavior system as a whole, ensuring that all components (IDE plugin, backend, AI model, vector database) function cohesively in a production-like environment. This level tests end-to-end workflows, such as code submission, analysis, and feedback delivery.

Approach:

- **Scope**: Test complete user workflows, including:
  - Student submitting Java code via the VS Code plugin and receiving feedback.
  - Instructor reviewing submissions and providing manual feedback.
  - Administrator configuring system settings (e.g., rate limits).
  - Error handling and recovery during submission or feedback generation.
- **Tools**: Selenium for UI testing, JMeter for simulating user interactions, and manual testing for qualitative validation.
- **Test Cases**: Cover functional scenarios (e.g., submitting valid code), non-functional requirements (e.g., response time under 2 seconds), and negative cases (e.g., submitting malformed code).

- **Execution**: Perform tests in a staging environment with VS Code, Spring Boot backend, and Milvus/ChromaDB deployed.
- **Metrics**: Achieve 95% pass rate for functional test cases and zero critical defects impacting core workflows.

Rationale: System testing ensures that the integrated system meets all functional and non-functional requirements, providing confidence in its readiness for deployment in an educational setting.

3.4. Performance Testing

Description: Performance testing evaluates the SyntaxSavior system's scalability, responsiveness, and stability under expected and peak loads, such as handling up to 50 concurrent submissions, as specified in the non-functional requirements.

Approach:

- **Scope**: Test system performance under:
    - Normal load (10–20 concurrent submissions).
    - Peak load (50 concurrent submissions, simulating a lab deadline).
    - Stress conditions (beyond 50 submissions to identify breaking points).
- **Tools**: JMeter for load testing, Grafana/Prometheus for monitoring server metrics (e.g., CPU, memory usage), and custom scripts for latency measurement.
- **Test Cases**: Measure response time for feedback generation (target: <2 seconds), throughput (submissions processed per minute), and resource utilization (e.g., database query latency).
- **Execution**: Conduct tests in a controlled environment with production-equivalent hardware and network configurations.
- **Metrics**: Ensure 99% of feedback responses are delivered within 2 seconds under peak load, with no system crashes or significant degradation.

Rationale: Performance testing validates the system's ability to handle concurrent users in a university setting, ensuring scalability and reliability during high-demand periods like assignment deadlines.

3.5. User Acceptance Testing (UAT)

Description: User Acceptance Testing (UAT) involves real users from the CMPE113 course (students, instructors, assistants) to validate the system's usability, educational effectiveness, and alignment with pedagogical goals. UAT ensures that SyntaxSavior meets stakeholder expectations in a real-world context.

Approach:

- **Scope**: Test usability of the IDE plugin (e.g., ease of code submission, clarity of feedback), feedback relevance, and instructor/administrator tools (e.g., grading, system configuration).
- **Participants**: 20–30 CMPE113 students, 2–3 instructors, and 2–3 lab assistants, representing all user roles.
- **Tools**: Surveys (Google Forms for feedback collection), usability testing frameworks (e.g., Morae for screen recording), and manual observation.
- **Test Cases**: Include tasks like submitting a lab assignment, reviewing AI feedback, and grading submissions, with qualitative feedback on clarity, helpfulness, and ease of use.
- **Execution**: Conduct UAT in a controlled lab setting with the production version of SyntaxSavior installed on university computers.
- **Metrics**: Achieve an average user satisfaction score of 4/5 or higher (based on survey responses) and address all critical usability issues raised.

Rationale: UAT ensures that SyntaxSavior is intuitive, educationally effective, and aligned with the needs of its primary users, fostering adoption and satisfaction in the CMPE113 course.

3.6. Beta Testing

Description: Beta testing involves a limited release of SyntaxSavior to selected CMPE113 lab sections to gather real-world feedback and identify issues before full deployment. This phase includes A/B testing to compare feature variations, such as automatic versus manual feedback triggers.

Approach:

- **Scope**: Test the system with 2–3 lab sections (approximately 50–75 students) over a 2-week period, focusing on stability, usability, and feature effectiveness.
- **Variations**: Conduct A/B testing for:
  - Automatic versus manual feedback triggers (e.g., real-time versus user-initiated analysis).
  - Feedback levels (e.g., detailed hints versus brief explanations).
- **Tools**: Bug tracking tools (e.g., Jira), telemetry for usage analytics (e.g., user interaction logs), and feedback forms for qualitative input.
- **Test Cases**: Include typical lab workflows (e.g., submitting assignments, reviewing feedback) and stress scenarios (e.g., multiple submissions during a lab session).
- **Execution**: Deploy SyntaxSavior in a production-like environment with monitoring and support for rapid issue resolution.
- **Metrics**: Achieve a bug-free experience for 90% of beta users, with actionable feedback to refine features before full rollout.

Rationale: Beta testing validates the system in a real-world academic setting, identifying usability and stability issues while optimizing features based on user preferences, ensuring a successful full deployment.

4. Test Environment

This section describes the hardware, software, and network configurations required for testing the SyntaxSavior system. The test environment is designed to replicate production conditions as closely as possible to ensure accurate and reliable test results, in accordance with IEEE 829 requirements.

Hardware:

- **Client Machines**: University lab computers with Intel Core i5/i3 processors, 8 GB RAM, and 512 GB SSD storage, running Windows 10/11 or Ubuntu 22.04.
- **Backend Server**: Dedicated server with 8-core CPU, 32 GB RAM, and 1 TB SSD, hosted on a cloud provider (e.g., AWS EC2 or Azure).
- **Database Server**: Separate instance for Milvus/ChromaDB with 4-core CPU, 16 GB RAM, and 500 GB SSD, optimized for vector search workloads.

Software:

- **IDE**: Visual Studio Code (version 1.85 or later) with the SyntaxSavior plugin installed.
- **Backend**: Spring Boot-based Java server (Java 17, Spring Boot 3.2), running on Apache Tomcat 10.
- **Database**: Milvus 2.3 or ChromaDB 0.4 for vector storage and retrieval, with PostgreSQL 16 for curriculum metadata.
- **AI Model**: Deepseek API endpoint (or equivalent fine-tuned model) hosted on a secure cloud service with REST API access.
- **Testing Tools**:

    - JUnit 5, Mockito, and JaCoCo for unit testing.
    - Postman and Selenium for integration testing.
    - JMeter and Grafana/Prometheus for performance testing.
    - Google Forms and Morae for UAT and beta testing feedback.
- **Monitoring**: Prometheus for server metrics, ELK Stack for log aggregation, and custom telemetry for usage analytics.

Network:

- **Configuration**: High-speed university LAN with 1 Gbps bandwidth, simulating production network conditions.
- **Security**: TLS 1.3 for encrypted communication, firewall rules to restrict access to authorized IPs, and VPN for remote testing if required.
- **Latency**: Target latency of <50 ms for internal network requests, with simulated external latency (100–200 ms) for cloud-based AI model access.

Environment Setup:

- **Staging Environment**: A fully configured replica of the production environment, including VS Code with the SyntaxSavior plugin, Spring Boot server, Milvus/ChromaDB, and Deepseek API integration, deployed on cloud infrastructure.
- **Test Data**: Synthetic and anonymized datasets mimicking CMPE113 lab assignments, including valid Java code, common errors (syntax, runtime, logical), and curriculum materials stored in the vector database.
- **Access Control**: Restricted access to the test environment, with credentials managed via a secure identity provider (e.g., university SSO or Keycloak).

Maintenance:

- **Version Control**: All software components tracked in a Git repository, with tagged releases for testing stability.
- **Environment Refresh**: Regular resets of the test environment to clear test data and ensure consistency, performed weekly or after major test cycles.
- **Documentation**: Detailed setup guides and configuration scripts maintained in the project repository to streamline environment provisioning.

Rationale: The test environment ensures that testing is conducted under realistic conditions, minimizing discrepancies between test results and production behavior. A well-defined environment supports repeatable and reliable tests, aligning with the system's quality and scalability objectives.

5. Sample Test Case

This section provides a sample test case for User Acceptance Testing (UAT) to validate the SyntaxSavior system's usability and educational effectiveness for CMPE113 students. The test case focuses on a typical student workflow—submitting a Java code assignment via the VS Code plugin and receiving AI-generated feedback—ensuring the system is intuitive and pedagogically sound.

Test Case ID: UAT-001
Title: Validate Student Code Submission and Feedback Retrieval
Feature Tested: IDE Plugin, Backend Processing, AI Intermediate Feedback, Feedback Accuracy
Test Objective: Verify that a CMPE113 student can submit a Java code assignment through the SyntaxSavior VS Code plugin, receive accurate and curriculum-aligned feedback, and find the process intuitive and helpful.
Test Type: User Acceptance Testing (Manual)
Preconditions:

- VS Code (version 1.85 or later) installed on a university lab computer with the SyntaxSavior plugin configured.

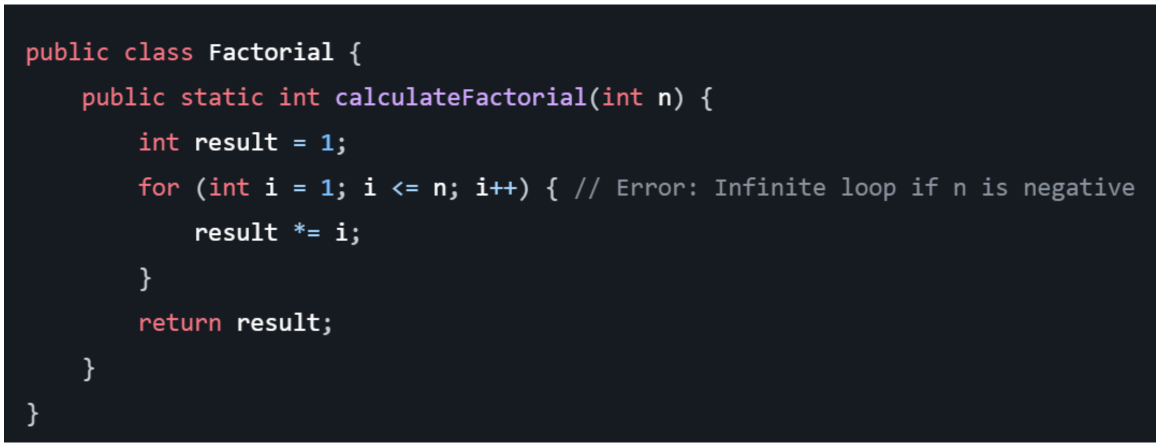
- Student user account created with valid credentials and CMPE113 course access.
- Test environment (staging) deployed with Spring Boot backend, Milvus/ChromaDB, and Deepseek API endpoint.
- Sample CMPE113 lab assignment (e.g., “Write a Java program to calculate the factorial of a number”) available in the curriculum database.

Test Steps:

1. Log in to the SyntaxSavior plugin using the student’s university credentials.
2. Open VS Code and create a new Java file for the lab assignment (e.g., Factorial.java).
3. Write a Java program with an intentional error (e.g., incorrect loop condition causing an infinite loop).
4. Trigger the SyntaxSavior plugin’s “Submit Code” feature (manual trigger via plugin UI).
5. Observe the plugin’s response, including error detection and feedback display.
6. Review the AI-generated feedback for clarity, relevance, and alignment with CMPE113 learning objectives.
7. Complete a post-test survey rating the usability (ease of submission, feedback clarity) and educational value (helpfulness of feedback) on a 1–5 scale.

Input Data:

- Java code with a logical error:



```java
public class Factorial {
    public static int calculateFactorial(int n) {
        int result = 1;
        for (int i = 1; i <= n; i++) { // Error: Infinite loop if n is negative
            result *= i;
        }
        return result;
    }
}
```

- Submission metadata: Student ID, Lab 1, CMPE113 course.

Expected Results:

- Step 1: Successful login with student role assigned.
- Step 4: Code submission completes within 2 seconds, with no errors in the plugin UI.
- Step 5: Plugin detects the logical error and displays a pop-up with feedback.
- Step 6: Feedback is clear, curriculum-aligned, and non-directive (e.g., "Check the loop condition to handle negative inputs, as factorial is undefined for negative numbers").
- Step 7: Student rates usability and educational value at 4/5 or higher in the survey, with no critical usability issues reported.

Actual Results: (To be recorded during testing)
Pass/Fail Criteria:

- Pass: All expected results are met, and student feedback score is ≥4/5.
- Fail: Any critical failure (e.g., submission error, irrelevant feedback, or score <4/5 with usability issues).

Test Environment:

- Hardware: University lab computer (Intel Core i5, 8 GB RAM, Windows 11).
- Software: VS Code with SyntaxSavior plugin, Spring Boot backend, Milvus/ChromaDB, Deepseek API.
- Network: University LAN with 1 Gbps bandwidth.

Roles and Responsibilities:

- **Test Executor**: CMPE113 student participant.
- **Test Facilitator**: QA engineer or lab assistant to guide the process and collect survey responses.
- **Test Analyst**: QA team member to review results and document findings.

Risks:

- Risk of unclear feedback confusing the student, mitigated by iterative feedback refinement during beta testing.
- Risk of plugin unresponsiveness, mitigated by performance testing and environment stability checks.

Rationale: This test case validates the core student workflow, ensuring that the system is user-friendly, delivers educationally effective feedback, and meets CMPE113 course requirements. UAT with real students provides critical insights into usability and pedagogical impact, aligning with the system's goals.

6. Conclusion

The Test Plan for the SyntaxSavior system provides a comprehensive framework for validating its functionality, performance, security, and educational effectiveness as an AI-powered programming education tool for CMPE113 at TED University. By systematically testing critical features—such as the IDE plugin, backend processing, AI feedback, user roles, and security mechanisms—the plan ensures that the system meets both functional and non-functional requirements outlined in the high-level and low-level design reports. The multi-level testing methodology, encompassing unit, integration, system, performance, user acceptance, and beta testing, guarantees thorough validation of all components and workflows, from code submission to feedback delivery.

Key objectives achieved through this test plan include verifying real-time code analysis, ensuring curriculum-aligned feedback, validating scalability under peak loads (e.g., 50 concurrent submissions), and confirming robust security against unauthorized access or malicious inputs. User acceptance testing with CMPE113 students, instructors, and assistants ensures that the system is intuitive and pedagogically sound, fostering independent learning while providing actionable guidance. Beta testing further refines the system by incorporating real-world feedback, preparing it for a successful full deployment.

Risks such as incorrect feedback, system overload, or security vulnerabilities are mitigated through rigorous testing and iterative improvements, aligning with IEEE 829 and engineering standards like IEEE 1016-2009 and SOLID principles. The test environment, replicating production conditions, ensures reliable and repeatable results, while clearly defined roles and responsibilities streamline execution. Ultimately, this test plan establishes SyntaxSavior as a reliable, secure, and effective tool for enhancing programming education, ready to support TED University's academic mission.

7. References

1. IEEE Computer Society. (2008). *IEEE Standard for Software and System Test Documentation (IEEE Std 829-2008)*. IEEE.
    - Provides the structure and guidelines for test plan documentation, used to ensure compliance in this report.
2. IEEE Computer Society. (2009). *IEEE Standard for Software and System Design Documentation (IEEE Std 1016-2009)*. IEEE.

- Referenced for aligning the test plan with design specifications and maintainability requirements.

3. International Organization for Standardization. (2018). *ISO/IEC/IEEE 29119-3:2013 - Software and Systems Engineering - Software Testing - Part 3: Test Documentation*. ISO/IEC/IEEE.

    - Guided the format and content of test cases and methodology descriptions.

4. Myers, G. J., Sandler, C., & Badgett, T. (2011). *The Art of Software Testing (3rd ed.)*. Wiley.

    - Informed testing methodologies, particularly unit, integration, and user acceptance testing approaches.

5. Sommerville, I. (2015). *Software Engineering (10th ed.)*. Pearson.

    - Provided insights into software testing best practices and performance testing strategies.

6. Spring Boot Documentation. (2023). *Spring Boot 3.2 Reference Guide*. Retrieved from https://docs.spring.io/spring-boot/docs/3.2.x/reference/html/.

    - Guided backend server setup and testing configurations.

7. Visual Studio Code Documentation. (2023). *VS Code Extension API*. Retrieved from https://code.visualstudio.com/api.

    - Referenced for plugin development and testing requirements.