System Verification and Validation Plan for Software Engineering

Team 4, EcoOptimizers

Nivetha Kuruparan
Sevhena Walker
Tanveer Brar
Mya Hussain
Ayushi Amin

March 11, 2025

Revision History

Date	Version	Notes
November 4th, 2024	0.0	Created initial revision of VnV Plan
March 10th, 2025	0.1	Revised Functional and Non-Functional Requirements

Contents

List of Tables iii

1 General Information 1
- 1.1 Summary 1
- 1.2 Objectives 1
- 1.3 Challenge Level and Extras 1
- 1.4 Relevant Documentation 2

2 Plan 2
- 2.1 Verification and Validation Team 2
- 2.2 SRS Verification Plan 3
- 2.3 Design Verification Plan 5
- 2.4 Verification and Validation Plan Verification Plan 7
- 2.5 Implementation Verification Plan 7
- 2.6 Automated Testing and Verification Tools 8
- 2.7 Software Validation Plan 9

3 System Tests 9
- 3.1 Tests for Functional Requirements 10
  - 3.1.1 Code Input Acceptance Tests 10
  - 3.1.2 Code Smell Detection Tests and Refactoring Suggestion (RS) Tests 11
  - 3.1.3 Output Validation Tests 11
  - 3.1.4 Tests for Reporting Functionality 12
  - 3.1.5 Documentation Availability Tests 13
  - 3.1.6 IDE Extension Tests 13
- 3.2 Tests for Nonfunctional Requirements 14
  - 3.2.1 Look and Feel 14
  - 3.2.2 Usability & Humanity 15
  - 3.2.3 Performance 18
  - 3.2.4 Operational & Environmental 18
  - 3.2.5 Maintenance and Support 19
  - 3.2.6 Security 22
  - 3.2.7 Cultural 23
  - 3.2.8 Compliance 24
- 3.3 Traceability Between Test Cases and Requirements 25

4 Unit Test Description 28
- 4.1 Detection Module 28
  - 4.1.1 Analyzer Controller 28
  - 4.1.2 String Concatenation in a Loop 29
  - 4.1.3 Long Element Chain 30
  - 4.1.4 Repeated Calls 31
  - 4.1.5 Long Message Chain 32

4.1.6 Long Lambda Element 34
4.2 CodeCarbon Measurement Module 35
4.3 Refactoring Module 36
4.3.1 Refactorer Controller 36
4.3.2 String Concatenation in a Loop 37
4.3.3 Long Element Chain 38
4.3.4 Member Ignoring Method 39
4.3.5 Use a Generator 40
4.3.6 Cache Repeated Calls 41
4.3.7 Long Parameter List 42
4.3.8 Long Message Chain 43
4.3.9 Long Lambda Element 44
4.4 VsCode Plugin 45
4.4.1 Detect Smells Command 45
4.4.2 Refactor Smells Command 46
4.4.3 Document Hashing 47
4.4.4 File Highlighter 48
4.4.5 Hover Manager 49
4.4.6 Line Selection Manager 50
4.4.7 Handle Smells Settings 51
4.4.8 Wipe Workspace Cache 52
4.4.9 Backend 53
4.5 API Routes 54
4.5.1 Smell Detection Endpoint 54
4.5.2 Refactoring Endpoint 55

References **56**

Appendices **57**

A Appendix **57**
A.1 Symbolic Parameters 57
A.2 Usability Survey Questions 57

B Reflection **58**

List of Tables

1 Functional Requirements and Corresponding Test Sections 25
2 Look & Feel Tests and Corresponding Requirements 25
3 Usability & Humanity Tests and Corresponding Requirements 26
4 Performance Tests and Corresponding Requirements 26
5 Operational & Environmental Tests and Corresponding Requirements 26
6 Maintenance & Support Tests and Corresponding Requirements 27
7 Security Tests and Corresponding Requirements 27

8 Cultural Tests and Corresponding Requirements 27
9 Compliance Tests and Corresponding Requirements 27

This document outlines the process and methods to ensure that the software meets its requirements and functions as intended. This document provides a structured approach to evaluating the product, incorporating both verification (to confirm that the software is built correctly) and validation (to confirm that the correct software has been built). By systematically identifying and mitigating potential issues, the V&V process aims to enhance quality, reduce risks, and ensure compliance with both functional and non-functional requirements.

The following sections will go over the approach for verification and validation, including the team structure, verification strategies at various stages and tools to be employed. Furthermore, a detailed list of system and unit tests are also included in this document.

1 General Information

1.1 Summary

The software being tested is called EcoOptimizer. EcoOptimizer is a python refactoring library that focuses on optimizing code in a way that reduces its energy consumption. The system will be capable to analyze python code in order to spot inefficiencies (code smells) within, measuring the energy efficiency of the inputted code and, of course, apply appropriate refactorings that preserve the initial function of the source code.

Furthermore, peripheral tools such as a Visual Studio Code (VS Code) extension and GitHub Action are also to be tested. The extension will integrate the library with Visual Studio Code for a more efficient development process and the GitHub Action will allow a proper integration of the library into continuous integration (CI) workflows.

1.2 Objectives

The primary objective of this project is to build confidence in the **correctness** and **energy efficiency** of the refactoring library, ensuring that it performs as expected in improving code efficiency while maintaining functionality. Usability is also emphasized, particularly in the user interfaces provided through the **VS Code extension** and **GitHub Action** integrations, as ease of use is critical for adoption by software developers. These qualities—correctness, energy efficiency, and usability—are central to the project's success, as they directly impact user experience, performance, and the sustainable benefits of the tool.

Certain objectives are intentionally left out-of-scope due to resource constraints. We will not independently verify external libraries or dependencies; instead, we assume they have been validated by their respective development teams.

1.3 Challenge Level and Extras

Our project, set at a **general** challenge level, includes two additional focuses: **user documentation** and **usability testing**. The user documentation aims to provide clear, accessible guidance for developers, making it easy to understand the tool's setup, functionality, and integration into existing workflows. Usability testing will ensure that the tool is intuitive and meets user needs effectively, offering insights to refine the user interface and optimize interactions with its features.

1.4 Relevant Documentation

The Verification and Validation (VnV) plan relies on three key documents to guide testing and assessment:

Software Requirements Specification (SRS) [7]: The foundation for the VnV plan, as it defines the functional and non-functional requirements the software must meet; aligning tests with these requirements ensures that the software performs as expected in terms of correctness, performance, and usability.

Module Interface Specification (MG) [5]: Provides detailed information about each module's interfaces, which is crucial for integration testing to verify that all modules interact correctly within the system.

Module Guide (MIS) [6]: Outlines the system's architectural design and module structure, ensuring the design of tests that align with the intended flow and dependencies within the system.

2 Plan

The following section outlines the comprehensive Verification and Validation (VnV) strategy, detailing the team structure, specific plans for verifying the Software Requirements Specification (SRS), design, implementation, and overall VnV process, as well as the automated tools employed and the approach to software validation.

2.1 Verification and Validation Team

The Verification and Validation (VnV) Team for the Source Code Optimizer project consists of the following members and their specific roles:

- **Sevhena Walker**: Lead Tester. Oversees and coordinates the testing process, ensuring all feedback is applied and all project goals are met.
- **Mya Hussain**: Functional Requirements Tester. Tests the software to verify that it meets all specified functional requirements.
- **Ayushi Amin**: Integration Tester. Focuses on testing the connection between the various components of the Python package, the VSCode plugin, and the GitHub Action to ensure seamless integration.
- **Tanveer Brar**: Non-Functional Requirements Tester. Assesses performance/security compliance with project standards.

- **Nivetha Kuruparan**: Non-Functional Requirements Tester. Ensures that the final product meets user expectations regarding user experience and interface intuitiveness.
- **Istvan David** (supervisor): Supervises the overall VnV process, providing feedback and guidance based on industry standards and practices.

2.2 SRS Verification Plan

Function & Non-Functional Requirements:

- A comprehensive test suite that covers all requirements specified in the SRS will be created.
- Each requirement will be mapped to specific test cases to ensure maximum coverage.
- Automated and manual testing will be conducted to verify that the implemented system meets each functional requirement.
- Usability testing with representative users will be carried out to validate user experience requirements and other non-functional requirements.
- Performance tests will be conducted to verify that the system meets specified performance requirements.

Traceability Matrix:

- We will create a requirements traceability matrix that links each SRS requirement to its corresponding implementation, test cases, and test results.
- This matrix will help identify any requirements that may have been overlooked during development.

Supervisor Review:

- After the implementation of the system, we will conduct a formal review session with key stakeholders such as our project supervisor, Dr. Istvan David.
- The stakeholders will be asked to verify that each requirement in the SRS is mapped out to specific expectations of the project.
- Prior to meeting, we will provide a summary of key requirements and design decisions and prepare a list specific questions or areas where we seek guidance.
- During the meeting, we will present an overview of the SRS using tables and other visual aids. We will conduct a walk through of critical section. Finally, we will discuss any potential risks or challenges identified.

User Acceptance Testing (UAT):

- We will involve potential end-users in testing the system to ensure it meets real-world usage scenarios.

- Feedback from UAT will be used to identify any discrepancies between the SRS and user expectations.

Continuous Verification:

- Throughout the development process, we will regularly review and update the SRS to ensure it remains aligned with the evolving system.
- Any changes to requirements will be documented and their impact on the system assessed.

Checklist for SRS Verification Plan

- ☐ Create comprehensive test suite covering all SRS requirements
- ☐ Map each requirement to specific test cases
- ☐ Conduct automated testing for functional requirements
- ☐ Perform manual testing for functional requirements
- ☐ Carry out usability testing with representative users
- ☐ Conduct performance tests to verify system meets requirements
- ☐ Create requirements traceability matrix
- ☐ Link each SRS requirement to implementation in traceability matrix
- ☐ Link each SRS requirement to test cases in traceability matrix
- ☐ Link each SRS requirement to test results in traceability matrix
- ☐ Schedule formal review session with project supervisor
- ☐ Prepare summary of key requirements and design decisions for supervisor review
- ☐ Prepare list of specific questions for supervisor review
- ☐ Create visual aids for SRS overview presentation
- ☐ Conduct walkthrough of critical SRS sections during review
- ☐ Discuss potential risks and challenges with supervisor
- ☐ Organize User Acceptance Testing (UAT) with potential end-users
- ☐ Collect and analyze UAT feedback
- ☐ Identify discrepancies between SRS and user expectations from UAT
- ☐ Establish process for regular SRS review and updates
- ☐ Document any changes to requirements
- ☐ Assess impact of requirement changes on the system

2.3 Design Verification Plan

Peer Review Plan:

- Each team member along with other classmates will thoroughly review the entire Design Document.
- A checklist-based approach will be used to ensure all key elements are covered.
- Feedback will be collected and discussed in a dedicated team meeting.

Supervisor Review:

- A structured review meeting will be scheduled with our project supervisor, Dr. Istvan David.
- We will present an overview of the design using visual aids (e.g., diagrams, tables).
- We will conduct a walkthrough of critical sections.
- We will use our project's issue tracker to document and follow up on any action items or changes resulting from this review.

☐ All functional requirements are mapped to specific design elements

☐ Each functional requirement is fully addressed by the design

☐ No functional requirements are overlooked or partially implemented

☐ Performance requirements are met by the design

☐ Scalability considerations are incorporated

☐ Reliability and availability requirements are satisfied

☐ Usability requirements are reflected in the user interface design

☐ High-level architecture is clearly defined

☐ Architectural decisions are justified with rationale

☐ Architecture aligns with project constraints and goals

☐ All major components are identified and described

☐ Interactions between components are clearly specified

☐ Component responsibilities are well-defined

☐ Appropriate data structures are chosen for each task

☐ Efficient algorithms are selected for critical operations

☐ Rationale for data structure and algorithm choices is provided

☐ UI design is consistent with usability requirements

☐ User flow is logical and efficient

☐ Accessibility considerations are incorporated

☐ All external interfaces are properly specified

☐ Interface protocols and data formats are defined

☐ Error handling for external interfaces is addressed

☐ Comprehensive error handling strategy is in place

☐ Exception scenarios are identified and managed

☐ Error messages are clear and actionable

☐ Authentication and authorization mechanisms are described

☐ Data encryption methods are specified where necessary

☐ Security best practices are followed in the design

☐ Design allows for future expansion and feature additions

☐ Code modularity and reusability are considered

☐ Documentation standards are established for maintainability

☐ Performance bottlenecks are identified and addressed

☐ Resource utilization is optimized

☐ Performance testing strategies are outlined

☐ Design adheres to established coding standards

☐ Industry best practices are followed

☐ Design patterns are appropriately applied

☐ All major design decisions are justified

☐ Trade-offs are explained with pros and cons

☐ Alternative approaches considered are documented

☐ Documents is clear, concise, and free of ambiguities

☐ Documents follows a logical structure

2.4 Verification and Validation Plan Verification Plan

The Verification and Validation (V&V) Plan for the Source Code Optimizer project serves as a critical document that requires a thorough examination to confirm its validity and effectiveness. To achieve this, the following strategies will be implemented:

1. **Peer Review**: Team members and peers will conduct a detailed review of the V&V plan. This process aims to uncover any gaps or areas that could benefit from enhancement, leveraging the collective insights of the group to strengthen the overall plan.
2. **Fault Injection Testing**: We will utilize mutation testing to assess the capability of our test cases to identify intentionally introduced faults. By generating variations of the original code, we can evaluate whether our testing strategies are robust enough to catch these discrepancies, hence enhancing the reliability of our verification process.
3. **Feedback Loop Integration**: Continuous feedback from review sessions and testing activities will be systematically integrated to refine the V&V plan. This ongoing process ensures the plan evolves based on insights gained from practical testing and peer input.

To comprehensively verify the V&V plan, we will utilize the following checklist:

- ☐ Does the V&V plan include all necessary aspects of software verification and validation?
- ☐ Are the roles and responsibilities clearly outlined within the V&V framework?
- ☐ Is there a diversity of testing methodologies included (e.g., unit testing, integration testing, system testing)?
- ☐ Does the plan have a clear process for incorporating feedback and gaining continuous improvement?
- ☐ Are success criteria established for each phase of testing?
- ☐ Is mutation testing considered to evaluate the effectiveness of the test cases?
- ☐ Are mechanisms in place to monitor and address any identified issues during the V&V process?
- ☐ Does the V&V plan align with the project timeline, available resources, and other constraints?

2.5 Implementation Verification Plan

The Implementation Verification Plan for the Source Code Optimizer project aims to ensure that the software implementation adheres to the requirements and design specifications defined in the SRS. Key components of this plan include:

- **Unit Testing**: A comprehensive suite of unit tests will be established to validate the functionality of individual components within the optimizer. These tests will specifically focus on the effectiveness of the code refactoring methods employed by the optimizer, utilizing `pytest` [2] for writing and executing these tests.

- **Static Code Analysis**: To maintain high code quality, static analysis tools will be employed. These tools will help identify potential bugs, security vulnerabilities, and adherence to coding standards in the Python codebase, ensuring that the optimizer is both efficient and secure.

- **Code Walkthroughs and Reviews**: The development team will hold regular code reviews and walkthrough sessions to collaboratively evaluate the implementation of the source code optimizer. These sessions will focus on code quality, readability, and compliance with the project's design patterns. Additionally, the final presentation will provide an opportunity for a thorough code walkthrough, allowing peers to contribute feedback on usability and functionality.

- **Continuous Integration**: The project will implement continuous integration practices using tools like GitHub Actions. This approach will automate the build and testing processes, allowing the team to verify that each change to the optimizer codebase meets the established quality criteria and integrates smoothly with the overall system.

- **Performance Testing**: The performance of the source code optimizer will be assessed to simulate various usage scenarios. This testing will focus on evaluating how effectively the optimizer processes large codebases and applies refactorings, ensuring that the tool operates efficiently under different workloads.

2.6 Automated Testing and Verification Tools

Unit Testing Framework: Pytest is chosen as the main framework for unit testing due to its (i) scalability (ii) integration with other tools (`pytest-cov` [4] for code coverage) (iii) extensive support for parameterized tests. These features make it easy to test the codebase as it grows, adapting to changes throughout the project's development.

Profiling Tool: The codebase will be evaluated based on results from both time and memory profiling to optimize computational speed and resource usage. For time profiling (recording the number of function calls, time spent in each function, and its descendants), `cProfile` will be used, as it is included within Python, making it a convenient choice for profiling. For memory profiling, `memory_profiler` [3] will be used, as it is easy to install and includes built-in support for visual display of output.

Static Analyzer: The codebase will be statically analyzed using `ruff` [1], as it provides fast linting, built-in rule enforcement, and integrates well with modern Python projects. `ruff` [1] enforces both linting and formatting rules, reducing the need for multiple tools.

Code Coverage Tools and Plan for Summary: The codebase will be analyzed to determine the percentage of code executed during tests. For granular-level coverage, `pytest-cov` [4] will be used, as it supports branch, line, and path coverage. Additionally, `pytest-cov` [4] integrates seamlessly with `pytest` [2], ensuring that test coverage results are generated alongside test execution.

Initially, the aim is to achieve a 40% coverage and gradually increment the level over time. Weekly reports generated from `pytest-cov` [4] will be used to track coverage trends and set goals accordingly to address any gaps in testing in the growing codebase.

Linters and Formatters: To enforce the official Python PEP 8 style guide and maintain code quality, the team will use `ruff` [1] for Python code and `eslint` [9] paired with `Prettier` [10] for the TypeScript extension.

Testing Strategy for the VSCode Extension: The TypeScript extension will be tested using `jest` [8]. Automated tests will verify interactions between the extension and the editor, reducing regressions during development.

CI Plan: As mentioned in the Development Plan, GitHub Actions will integrate the above tools within the CI pipeline. GitHub Actions will be configured to run unit tests written in `pytest`, perform static analysis using `ruff` [1], and execute `pytest-cov` [4] for test coverage. For the TypeScript extension, a pre-commit will run `eslint` [9] and `Prettier` [10], and automated tests will be executed as a GitHub Action using `jest` [8]. Through automated testing, any errors, code style violations, and regressions will be promptly identified.

2.7 Software Validation Plan

- One or more open source Python code bases will be used to test the tool on. Based on its performance in functional and non-functional tests outlined in further sections of the document, the software can be validated against defined requirements.
- In addition to this, the team will reach out to Dr David as well as a group of volunteer Python developers to perform usability testing on the IDE plugin workflow as well as the CI/CD workflow.
- The team will conduct a comprehensive review of the requirements from Dr David through the Rev 0 Demo.

3 System Tests

This section outlines the tests for verifying both functional and nonfunctional requirements of the software, ensuring it meets user expectations and performs reliably. This includes tests for code quality, usability, performance, security, and traceability, covering essential aspects of the software's operation and compliance.

3.1 Tests for Functional Requirements

The subsections below outline tests corresponding to functional requirements in the SRS [7]. Each test is associated with a unique functional area, helping to confirm that the tool meets the specified requirements. Each functional area has its own subsection for clarity.

3.1.1 Code Input Acceptance Tests

This section covers the tests for ensuring the system correctly accepts Python source code files, detects errors in invalid files, and provides suitable feedback (FR 1).

test-FR-IA-1 Valid Python File Acceptance

Control: Manual
Initial State: Tool is idle.
Input: A valid Python file (filename.py) with valid standard syntax.
Output: The system accepts the file without errors.

Test Case Derivation: Confirming that the system correctly processes a valid Python file as per FR 1.

How test will be performed: Feed a syntactically valid .py file to the tool and observe if it's accepted without issues.

test-FR-IA-2 Feedback for Python File with Bad Syntax

Control: Manual
Initial State: Tool is idle.
Input: A .py file (badSyntax.py) containing deliberate syntax errors that render the file unrunnable.
Output: The system rejects the file and provides an error message detailing the syntax issue.

Test Case Derivation: Verifies the tool's handling of syntactically invalid Python files to ensure user awareness of the syntax issue, meeting FR 1.

How test will be performed: Feed a .py file with syntax errors to the tool and check that the system identifies it as invalid and produces an appropriate error message.

test-FR-IA-3 Feedback for Non-Python File

Control: Manual
Initial State: Tool is idle.
Input: A non-Python file (document.txt) or a file with an incorrect extension (script.js).
Output: The system rejects the file and provides an error message indicating the invalid file format.

Test Case Derivation: Ensures the tool detects unsupported file types and provides feedback, satisfying FR 1.

How test will be performed: Attempt to load a .txt or other non-Python file, and verify that the system rejects it with a message indicating an invalid file type.

3.1.2 Code Smell Detection Tests and Refactoring Suggestion (RS) Tests

This area includes tests to verify the detection and refactoring of specified code smells that impact energy efficiency (FR 2). These tests will be done through unit testing.

3.1.3 Output Validation Tests

The following tests are designed to validate that the functionality of the original Python code remains intact after refactoring. Each test ensures that the refactored code passes the same test suite as the original code, confirming compliance with functional requirement FR 3.

test-FR-OV-1 Verification of Valid Python Output

Control: Manual
Initial State: Tool has processed a file with detected code smells.
Input: Output refactored Python code.
Output: Refactored code is syntactically correct and Python-compliant.

Test Case Derivation: Ensures refactored code remains valid and usable, satisfying FR 6.

How test will be performed: Run a linter on the output code and verify it passes without syntax errors.

3.1.4 Tests for Reporting Functionality

The reporting functionality of the tool is crucial for providing users with comprehensive insights into the refactoring process, including detected code smells, refactorings applied, energy consumption measurements, and the results of the original test suite. This section outlines tests that ensure the reporting feature operates correctly and delivers accurate, well-structured information as specified in the functional requirements (FR 9).

test-FR-RP-1 A Report With All Components Is Generated

Control: Manual **Initial State:** The tool has completed refactoring a Python code file.
Input: The refactoring results, including detected code smells, applied refactorings, and energy consumption metrics.
Output: A well-structured report is generated, summarizing the refactoring process.

Test Case Derivation: This test ensures that the tool generates a comprehensive report that includes all necessary information as required by FR 9.

How test will be performed: After refactoring, the tool will invoke the report generation feature and a user can validate that the output meets the structure and content specifications.

test-FR-RP-2 Validation of Code Smell and Refactoring Data in Report

Control: Automatic
Initial State: The tool has identified code smells and performed refactorings.
Input: The results of the refactoring process.
Output: The generated report accurately lists all detected code smells and the corresponding refactorings applied.

Test Case Derivation: This test verifies that the report includes correct and complete information about code smells and refactorings, in compliance with FR 9.

How test will be performed: The tool will compare the contents of the generated report against the detected code smells and refactorings to ensure accuracy.

test-FR-RP-3 Energy Consumption Metrics Included in Report

Control: Manual **Initial State:** The tool has measured energy consumption before and after refactoring.
Input: Energy consumption metrics obtained during the refactoring process.
Output: The report presents a clear comparison of energy usage before and after the refactorings.

Test Case Derivation: This test confirms that the reporting feature effectively communicates energy consumption improvements, aligning with FR 9.

How test will be performed: A user will analyze the energy metrics in the report to ensure they accurately reflect the measurements taken during the refactoring.

test-FR-RP-4 Functionality Test Results Included in Report

Control: Automatic
Initial State: The original test suite has been executed against the refactored code.
Input: The outcomes of the test suite execution.
Output: The report summarizes the test results, indicating which tests passed and failed.

Test Case Derivation: This test ensures that the reporting functionality accurately reflects the results of the test suite as specified in FR 9.

How test will be performed: The tool will generate the report and validate that it contains a summary of test results consistent with the actual test outcomes.

3.1.5 Documentation Availability Tests

The following test is designed to ensure the availability of documentation as per FR 10.

test-FR-DA-1 Test for Documentation Availability

Control: Manual
Initial State: The system may or may not be installed.
Input: User attempts to access the documentation.
Output: The documentation is available and covers installation, usage, and troubleshooting.

Test Case Derivation: Validates that the documentation meets user needs (FR 10).

How test will be performed: Review the documentation for completeness and clarity.

3.1.6 IDE Extension Tests

The following tests are designed to ensure that the user can integrate the tool into VS Code IDE as specified in FR 11 and that the tool works as intended as an extension.

test-FR-IE-1 Installation of Extension in Visual Studio Code

Control: Manual
Initial State: The user has Visual Studio Code installed on their machine.
Input: The user attempts to install the refactoring tool extension from the Visual Studio Code Marketplace.
Output: The extension installs successfully, and the user is able to see it listed in the Extensions view.

Test Case Derivation: This test validates the installation process of the extension to ensure that users can easily add the tool to their development environment.

How test will be performed:

1. Open Visual Studio Code.
2. Navigate to the Extensions view (Ctrl+Shift+X).
3. Search for the refactoring tool extension in the marketplace.
4. Click on the "Install" button.
5. After installation, verify that the extension appears in the installed extensions list.
6. Confirm that the extension is enabled and ready for use by checking its functionality within the editor.

test-FR-IE-2 Running the Extension in Visual Studio Code

Control: Manual
Initial State: The user has successfully installed the refactoring tool extension in Visual Studio Code.
Input: The user opens a Python file and activates the refactoring tool extension.
Output: The extension runs successfully, and the user can see a list of detected code smells and suggested refactorings.

Test Case Derivation: This test validates that the extension can be executed within the development environment and that it correctly identifies code smells as per the functional requirements in the SRS.

How test will be performed:

1. Open Visual Studio Code.
2. Open a valid Python file that contains known code smells.
3. Activate the refactoring tool extension using the command palette (Ctrl+Shift+P) and selecting the extension command.
4. Observe the output panel for the detection of code smells.
5. Verify that the extension lists the identified code smells and provides appropriate refactoring suggestions.
6. Confirm that the suggestions are relevant and feasible for the detected code smells.

3.2 Tests for Nonfunctional Requirements

The section will cover system tests for the non-functional requirements (NFR) listed in the SRS document[7]. The goal for these tests is to address the fit criteria for the requirements. Each test will be linked back to a specific NFR that can be observed in section 3.3.

3.2.1 Look and Feel

The following subsection tests cover all Look and Feel requirements listed in the SRS [7]. They seek to validate that the system is modern, visually appealing, and supporting of a calm and focused user experience.

test-LF-1 Side-by-side code comparison in IDE plugin

Type: Non-Functional, Manual, Dynamic
Initial State: IDE plugin open in VS Code, with a sample code file loaded
Input/Condition: The user initiates a refactoring operation
Output/Result: The plugin displays the original and refactored code side by side

How test will be performed: The tester will open a sample code file within the IDE plugin and apply a refactoring operation. After refactoring, they will verify that the original code appears on one side of the interface and the refactored code on the other, with clear options to accept or reject each change. The tester will interact with the accept/reject buttons to ensure functionality and usability, confirming that users can seamlessly make refactoring decisions with both versions displayed side by side.

test-LF-2 Theme adaptation in VS Code

Type: Non-functional, Manual, Dynamic
Initial State: IDE plugin open in VS Code with either light or dark theme enabled
Input/Condition: The user switches between light and dark themes in VS Code
Output/Result: The plugin's interface adjusts automatically to match the theme

How test will be performed: The tester will open the plugin in both light and dark themes within VS Code by toggling the theme settings in the IDE. They will observe the plugin interface each time the theme is switched, ensuring that the plugin automatically adjusts to match the selected theme without any manual adjustments required.

test-LF-3 Design Acceptance

Type: Non-Functional, Manual, Dynamic
Initial State: IDE plugin open
Input/Condition: User interacts with the plugin
Output/Result: A survey report

How test will be performed: After a testing session, developers fill out the survey found in A.2 evaluating their experience with the plugin.

3.2.2 Usability & Humanity

The following subsection tests cover all Usability & Humanity requirements listed in the SRS [7]. They seek to validate that the system is accessible, user-centred, intuitive and easy to navigate.

test-UH-1 Customizable settings for refactoring preferences

Type: Non-Functional, Manual, Dynamic
Initial State: IDE plugin open with settings panel accessible
Input/Condition: User customizes refactoring style and detection sensitivity

Output/Result: Custom configurations save and load successfully

How test will be performed: The tester will navigate to the settings menu within the tool and adjust various options, including refactoring style, colour-coded indicators, and unit preferences (metric vs. imperial). After each adjustment, the tester will observe if the interface and refactoring suggestions reflect the changes made.

test-UH-2 Multilingual support in user guide

Type: Non-Functional, Manual, Dynamic
Initial State: Bilingual user navigates to system documentation
Input/Condition: User accesses guide in both English and French
Output/Result: The guide is accessible in both languages

How test will be performed: The tester will set the tool's language to French and access the user guide, reviewing each section to ensure accurate translation and readability. After verifying the French version, they will switch the language to English, confirming consistency in content, layout, and clarity between both versions.

test-UH-3 YouTube installation tutorial availability

Type: Non-Functional, Manual, Dynamic
Initial State: User access documentation resources
Input/Condition: User follows the provided link to a YouTube tutorial
Output/Result: Installation tutorial is available and accessible on YouTube, and user successfully installs the system.

How test will be performed: The tester will start with the installation instructions provided in the user guide and follow the link to the YouTube installation tutorial. They will watch the video and proceed with each installation step as demonstrated. Throughout the process, the tester will note the clarity and pacing of the instructions, any gaps between the video and the actual steps, and if the video effectively guides them to a successful installation.

test-UH-4 High-Contrast Theme Accessibility Check

Objective: Evaluate the high-contrast themes in the refactoring tool for compliance with accessibility standards to ensure usability for visually impaired users.
Scope: Focus on UI components that utilize high-contrast themes, including text, buttons, and backgrounds.
Methodology: Static Analysis
Process:

- Identify all colour codes used in the system and categorize them by their role in the UI (i.e. background, foreground text, buttons, etc.).
- Use tools to measure colour contrast ratios against WCAG thresholds (4.5:1 for normal text, 3:1 for large text [11].

Roles and Responsibilities: Developers implement themes that pass the testing process.

Tools and Resources: WebAIM Color Contrast Checker, WCAG guidelines documentation, internal coding standards.

Acceptance Criteria: All UI elements must meet WCAG contrast ratios; documentation must accurately reflect theme usage.

test-UH-5 Intuitive user interface for core functionality

Type: Non-Functional, User Testing, Dynamic
Initial State: IDE plugin open with code loaded
Input/Condition: User interacts with the plugin
Output/Result: Users can access core functions within three clicks or less

How test will be performed: After a testing session, developers fill out the survey found in A.2 evaluating their experience with the plugin.

test-UH-6 Clear and concise user prompts

Type: Non-Functional, User Survey, Dynamic
Initial State: IDE plugin prompts user for input
Input/Condition: Users follow on-screen instructions
Output/Result: 90% of users report the prompts are straightforward and effective

How test will be performed: Users complete tasks requiring prompts and answer the survey found in A.2 on the clarity of guidance provided.

test-UH-7 Context-sensitive help based on user actions

Type: Non-Functional, Manual, Dynamic
Initial State: IDE plugin open with help function enabled
Input/Condition: User engages in various actions, requiring guidance
Output/Result: Help resources are accessible within 1-3 clicks

How test will be performed: The tester will perform a series of tasks within the tool, such as initiating a code analysis, applying a refactoring, and adjusting settings. At each step, they will access the context-sensitive help option to confirm that the information provided is relevant to the current task. The tester will evaluate the ease of accessing help, the relevance and clarity of guidance, and whether the help content effectively supports task completion.

test-UH-8 Clear and constructive error messaging

Type: Non-Functional, Manual, Dynamic
Initial State: IDE plugin open with possible error scenarios triggered
Input/Condition: User encounters an error during use
Output/Result: 80% of users report that error messages are helpful and courteous

How test will be performed: After receiving error messages, users fill out the survey found in A.2 on their clarity and constructiveness.

3.2.3 Performance

The following subsection tests cover all Performance requirements listed in the SRS [7]. These tests validate the tool's efficiency and responsiveness under varying workloads, including code analysis, refactoring, and data reporting.

test-PF-1 Performance and capacity validation for analysis and refactoring

Type: Non-Functional, Automated, Dynamic
Initial State: IDE open with multiple python files of varying sizes ready (250, 1000, 3000 lines of code).
Input/Condition: Initiate the refactoring process for each file sequentially
Output/Result: Process completes within 20 seconds for files up to 250 lines of code, 50 seconds for 1000 lines of code and within 2 minutes for 3000 lines of code.

How test will be performed: The tester will use three python files of different sizes: small (250 lines), medium (1000 lines), and large (3000 lines). For each file, start the refactoring process while running a timer. The scope of the test ends when the system presents the user with the completed refactoring proposal. The time taken for the total detection + refactoring is checked against the expected result.

test-PF-2 Accuracy of code smell detection

Type: Non-Functional, Automated, Dynamic
Initial State: Python file containing pre-determined code smells ready for refactoring with proper configurations for the system
Input/Condition: User initiates refactoring on the code file
Output/Result: All code smells determined prior to the test are detected.

How test will be performed: see tests in the Code Smell Detection section.

test-PF-3 Valid syntax and structure in refactored code

Type: Non-Functional, Automated, Dynamic
Initial State: A refactored code file is present in the user's workspace
Input/Condition: A python linter is run on the refactored python file
Output/Result: Refactored code meets Python syntax and structural standards

How test will be performed: see test test-FR-OV-2

3.2.4 Operational & Environmental

The following subsection tests cover all Operational and Environmental requirements listed in the SRS [7]. Testing includes adherence to emissions standards, integration with environmental metrics, and adaptability to diverse operational settings.

test-OPE-1 VS Code compatibility for refactoring library extension

Type: Non-Functional, Manual, Dynamic
Initial State: VS Code IDE open and library installed
Input/Condition: User installs and opens the refactoring library extension in VS Code
Output/Result: The refactoring library extension installs successfully and runs within VS Code

How test will be performed: The tester will navigate to the VS Code marketplace, search for the refactoring library extension, and install it. Once installed, the tester will open the extension and perform a basic refactoring task to ensure the tool operates correctly within the VS Code environment and has access to the system library.

test-OPE-2 Import and export capabilities for codebases and metrics

Type: Non-Functional, Manual, Dynamic
Initial State: IDE plugin open with the option to import/export codebases and metrics
Input/Condition: User imports an existing codebase and exports refactored code and metrics reports
Output/Result: The tool successfully imports codebases, refactors them, and exports both code and metrics reports

How test will be performed: The tester will load an existing codebase into the tool, initiate refactoring, and select the option to export the refactored code and metrics report. The export should generate files in the selected format. The tester will verify the file formats, check for correct data structure, and validate that the content accurately reflects the refactoring and metrics generated by the tool.

test-OPE-3 PIP package installation availability

Type: Non-Functional, Manual, Dynamic
Initial State: Python environment ready without the refactoring library installed
Input/Condition: User installs the refactoring library using the command `pip install ecooptimizer`
Output/Result: The library installs successfully without errors and is available for use in Python scripts

How test will be performed: The tester will open a new Python environment and enter the command to install the refactoring library via PIP. Once installed, the tester will import the library in a Python script and execute a basic function to confirm successful installation and functionality. The test verifies the library's availability and ease of installation for end users.

3.2.5 Maintenance and Support

The following subsection tests cover all Maintenance and Support requirements listed in the SRS [7]. These tests focus on rollback capabilities, compatibility with external libraries, automated testing, and extensibility for adding new code smells and refactoring functions.

test-MS-1 Extensibility for New Code Smells and Refactorings

Objective: Confirm that the tool's architecture allows for the addition of new code smell detections and refactoring techniques with minimal code changes and disruption to existing functionality.

Scope: This test applies to the tool's extensibility, including modularity of code structure, ease of integration for new detection methods, and support for customization.

Methodology: Code walkthrough

Process:

- Conduct a code walkthrough focusing on the modularity and structure of the code smell detection and refactoring components.
- Add a sample code smell detection and refactoring function to validate the ease of integration within the existing architecture.
- Verify that the new function integrates seamlessly without altering existing features and that it is accessible through the tool's main interface.

Roles and Responsibilities: Once the system is complete, the development team will perform the code walkthrough and integration. They will review and approve any structural changes required.

Tools and Resources: Code editor, tool's developer documentation, sample code smell and refactoring patterns

Acceptance Criteria: New code smells and refactoring functions can be added within the existing modular structure, requiring minimal changes. The new function does not impact the performance or functionality of existing features.

test-MS-2 Maintainable and Adaptable Codebase

Objective: Ensure that the codebase is modular, well-documented, and maintainable, supporting future updates and adaptations for new Python versions and standards.

Scope: This test covers the maintainability of the codebase, including structure, documentation, and modularity of key components.

Methodology: Static analysis and documentation walkthrough

Process:

- Review the codebase to verify the modular organization and clear separation of concerns between components.
- Examine documentation for code clarity and completeness, especially around key functions and configuration files.

- Assess code comments and the quality of function/method naming conventions, ensuring readability and consistency for future maintenance.

Roles and Responsibilities: Once the system is complete, the development team will conduct the code review, to identify areas for improvement. If necessary, they will also ensure to improve the quality of the documentation.

Tools and Resources: Code editor, documentation templates, code commenting standards, Python development guides

Acceptance Criteria: The codebase is modular and maintainable, with sufficient documentation to support future development. All major components are organized to allow for easy updates with minimal impact on existing functionality.

test-MS-3 Easy rollback of updates in case of errors

Type: Non-Functional, Manual, Dynamic
Initial State: Latest version of the tool installed with the ability to apply and revert updates
Input/Condition: User applies a simulated new update and initiates a rollback
Output/Result: The system reverts to the previous stable state without any errors

How test will be performed: The tester will apply a simulated update. Following this, they will initiate the rollback function, which should restore the tool to its previous stable version. The tester will verify that all features function as expected post-rollback and document the time taken to complete the rollback process

3.2.6 Security

The following subsection tests cover all Security requirements listed in the SRS [7]. These tests seek to validate that the tool is protected against unauthorized access, data breaches, and external threats.

test-SRT-1 Audit Logs for Refactoring Processes

Objective: Ensure that the tool maintains a secure, tamper-proof log of all refactoring processes, including pattern analysis, energy analysis, and report generation, for accountability in refactoring events.

Scope: This test covers the logging of refactoring events, ensuring logs are complete and tamper-proof for future auditing needs.

Methodology: Code walkthrough and static analysis

Process:

- Review the codebase to confirm that each refactoring event (e.g., pattern analysis, energy analysis, report generation) is logged with details such as timestamps and event descriptions.
- Document any logging gaps or security vulnerabilities, and consult with the development team to implement enhancements.

Roles and Responsibilities: The development team will review and test the logging mechanisms, with the project supervisor ensuring alignment with auditing requirements.

Tools and Resources: Access to logging components, tamper-proof logging tools

Acceptance Criteria: All refactoring processes are logged in a secure, tamper-proof manner, ensuring complete traceability for future audits.

3.2.7 Cultural

Cultural requirements are not applicable to this project since we are using VS Code settings and a plugin-based approach. These aspects do not involve cultural considerations, making such requirements unnecessary.

3.2.8 Compliance

The following subsection tests cover all Compliance requirements listed in the SRS [7]. The tests focus on adherence to PIPEDA, CASL, and ISO 9001, as well as SSADM standards, ensuring the tool complies with relevant regulations and aligns with professional development practices.

test-CPL-1 Compliance with PIPEDA and CASL

Objective: Ensure the tool's data handling and communication practices align with the Personal Information Protection and Electronic Documents Act (PIPEDA) and Canada's Anti-Spam Legislation (CASL). The focus is on ensuring compliance through best practices rather than direct data storage verification as no data is locally stored.

Scope: This test applies to all processes related to data handling and user communication to verify compliance with PIPEDA and CASL.

Methodology: Comparison of code data handling processes with compliance best practices

Process:

- Review the tool's data handling procedures to confirm all processing remains local and that no user data is stored.
- Verify that no personal data collection occurs, ensuring no explicit user consent is required beyond general software disclaimers.
- Inspect communication practices to ensure compliance with CASL, confirming that the tool does not send unsolicited communications.

Roles and Responsibilities: The development team will review compliance best practices and update documentation as needed.

Tools and Resources: Access to PIPEDA and CASL guidelines

Acceptance Criteria: Compliance is confirmed if no unsolicited communications occur and all best practices are followed. The tool must ensure that all data handling remains local and that no user data is stored or transmitted externally.

test-CPL-2 Compliance with ISO 9001 and SSADM Standards

Objective: Assess whether the tool's quality management and software development processes follow structured methodologies in line with ISO 9001 quality management principles and SSADM (Structured Systems Analysis and Design Method) best practices.

Scope: This test evaluates development workflows, documentation practices, and adherence to structured methodologies.

Methodology: Documentation review and evaluation of structured development practices

Process:

- Review development documentation to ensure structured workflow practices are followed.
- Evaluate version control and change tracking methods to confirm basic quality assurance measures exist.
- Identify any gaps in structured development adherence and suggest improvements.
- Validate improvements through informal documentation and process reviews.

Roles and Responsibilities: The development team will conduct an internal workflow assessment, and updates will be made to improve documentation and structured processes.

Tools and Resources: Development documentation, version control records, best practice comparisons

Acceptance Criteria: The tool's development must follow a structured approach with version control, clear documentation, and logical workflow practices. Compliance is met if basic structured development principles are followed, even without formal certification.

3.3 Traceability Between Test Cases and Requirements

Table 1: Functional Requirements and Corresponding Test Sections

Section	**Functional Requirement**
Input Acceptance Tests	FR 1
Code Smell Detection Tests	FR 2
Refactoring Suggestion Tests	FR 4
Output Validation Tests	FR 3, FR 6
Tests for Report Generation	FR 9
Documentation Availability Tests	FR 10
IDE Integration Tests	FR 11

Table 2: Look & Feel Tests and Corresponding Requirements

Test ID (test-)	**Non-Functional Requirement**
LF-1	LFR-AP 1
LF-2	LFR-AP 2
LF-3	LFR-AP 3
LF-4	LFR-AP 5
LF-5	LFR-AP 4, LFR-ST 1-3

Table 3: Usability & Humanity Tests and Corresponding Requirements

Test ID (test-)	**Non-Functional Requirement**
UH-1	UHR-PS1 1
UH-2	UHR-PS1 2, MS-SP 1
UH-3	UHR-LRN 2
UH-4	UHR-ACS 1
UH-5	UHR-ACS 2
UH-6	UHR-EOU 1
UH-7	UHR-EOU 2
UH-8	UHR-LRN 1
UH-9	UHR-UPL 1

Table 4: Performance Tests and Corresponding Requirements

Test ID (test-)	**Non-Functional Requirement**
PF-1	PR-SL 1, PR-SL 2, PR-CR 1
PF-2	PR-SCR 1
PF-3	PR-PAR 1
PF-4	PR-PAR 2
PF-5	PR-PAR 3
PF-6	PR-RFT 1
PF-7	PR-RFT 2
PF-8	PR-LR 1, MS-MNT 5

Table 5: Operational & Environmental Tests and Corresponding Requirements

Test ID (test-)	**Non-Functional Requirement**
Not explicitly tested	OER-EP 1
Not explicitly tested	OER-EP 2
OPE-1	OER-WE 1
OPE-2	OER-IAS 1
OPE-3	OER-IAS 2
OPE-4	OER-IAS 3
OPE-5	OER-PR 1
Tested by FRs	OER-RL 1
Not explicitly tested	OER-RL 2

Table 6: Maintenance & Support Tests and Corresponding Requirements

Test ID (test-)	**Non-Functional Requirement**
MS-1	MS-MNT 1, PR-SER 1
MS-2	MS-MNT 2
MS-3	MS-MNT 3
Not explicitly tested	MS-MNT 4

Table 7: Security Tests and Corresponding Requirements

Test ID (test-)	**Non-Functional Requirement**
SRT-1	SR-AR 1
SRT-2	SR-AR 2
SRT-3	SR-IR 1
SRT-4	SR-PR 1
SRT-5	SR-PR 2
SRT-6	SR-AUR 1
SRT-7	SR-AUR 2
SRT-8	SR-IM 1

Table 8: Cultural Tests and Corresponding Requirements

Test ID (test-)	**Non-Functional Requirement**
CULT-1	CULT 1
CULT-2	CULT 2
CULT-3	CULT 3

Table 9: Compliance Tests and Corresponding Requirements

Test ID (test-)	**Non-Functional Requirement**
CPL-1	CL-LR 1
CPL-2	CL-SCR 1

4 Unit Test Description

4.1 Detection Module

4.1.1 Analyzer Controller

Goal: The analyzer controller serves as the central coordination unit for detecting code smells using various analysis methods. It interfaces with multiple analyzers, processes detection results, and ensures accurate filtering and customization of analysis options. The following unit tests verify the correctness of this functionality.

The tests validate the proper execution of smell detection, correct filtering of smells based on the chosen analysis method, appropriate handling of missing or disabled smells, and the generation of custom analysis options. Edge cases such as empty files, incorrect configurations, and mismatched smell detection methods are also considered.

Target requirement(s): FR2, FR5 [7]

- **Running Smell Detection Analysis**
  - Ensures the analyzer correctly detects CRC smells in a given Python file.
  - Validates that detected smells include the correct attributes, such as message ID, file path, and confidence level.
  - Confirms that logs capture detected smells for debugging.
- **Handling Cases with No Detected Smells**
  - Ensures that when no smells are detected, the controller logs an appropriate success message.
  - Verifies that the returned list of smells is empty.
- **Filtering Smells by Analysis Method**
  - Ensures that the controller correctly filters smells based on the specified analysis method (e.g., AST, Astroid, Pylint).
  - Verifies that only smells matching the given method are returned.
- **Generating Custom Analysis Options**
  - Ensures that the controller correctly generates custom options for AST and Astroid analysis.
  - Validates that generated options include callable detection functions for applicable smells.

The test cases for this module can be found here.

4.1.2 String Concatenation in a Loop

Goal: The string concatenation in a loop detection module identifies inefficient string operations occurring inside loops, which can significantly impact performance. The following unit tests verify the detection capabilities.

The tests evaluate different types of string concatenation scenarios within loops, ensuring that various cases are detected correctly. These include basic concatenation, nested loops, conditional concatenation, type inference, and different string interpolation methods. The robustness of the detection logic is assessed through multiple test cases covering common patterns.

Target requirement(s): FR2 [7]

- **Basic Concatenation in a Loop**
  - Simple assignment string concatenation of the form `var = var +` ... inside `for` and `while` loops.
  - Augmented assignment string concatenation using `+=` inside `for` and `while` loops.
  - Concatenation involving object attributes (e.g., `var.attr +=` ...).
  - Concatenation modifying values inside complex objects (i.e., dictionaries, iterables).
- **Nested Loop Concatenation**
  - String concatenation inside nested loops where the outer loop contributes to the concatenation.
  - Resetting the concatenation variable inside the outer loop but continuing inside the inner loop.
- **Conditional Concatenation**
  - String concatenation inside loops with `if-else` conditions modifying the concatenation variable.
  - Different branches of a condition appending different string literals.
- **Type Inference**
  - Proper detection of string types through initial variable assignment.
  - Proper detection of string types through assignment type hints.
  - Proper detection of string types through function definition type hints.
  - Proper detection of string types initialized as class variables.
- **String Interpolation Methods**

  - Concatenation using `%` formatting.
  - Concatenation using `str.format()`.
  - Concatenation using f-strings inside loops.

- **Concatenation with Repeated Reassignment**
  - A variable being reassigned multiple times in the same loop iteration before proceeding to the next iteration.

- **Concatenation with Variable Access Inside the Loop**
  - Cases where the concatenation variable is accessed inside the loop, making refactoring ineffective since the joined result would be required at every iteration.

- **Concatenation Order Sensitivity**
  - Concatenation where new values are inserted at the beginning of the string instead of the end.
  - Concatenation that involves both prefix and suffix additions within the same loop.

The test cases for this module can be found here.

4.1.3 Long Element Chain

Goal: The long element chain detection module identifies excessive dictionary access chains that exceed a predefined threshold (default: 3). The following unit tests verify the detection capabilities.

The tests evaluate the detection of long element chains by verifying that sequences exceeding the threshold, such as deep dictionary accesses, are correctly identified. They include edge cases, such as chains just below the threshold, variations in data structures, and multiple chains in the same scope, to assess the robustness of the detection logic.

Target requirement(s): FR2 [7]

- **Ignores code with no chains**
  - Ensures that code without any nested element chains does not trigger a detection.

- **Ignores chains below the threshold**
  - Verifies that element chains shorter than the threshold are not flagged.

- **Detects chains exactly at threshold**
  - Ensures that an element chain with a length equal to the threshold is flagged.

- **Detects chains exceeding the threshold**
  - Verifies that chains longer than the threshold are correctly detected.
- **Detects multiple chains in the same file**
  - Ensures that multiple long element chains appearing in different locations within the same code file are individually detected.
- **Detects chains inside nested functions and classes**
  - Confirms that element chains occurring within functions and class methods are correctly identified.
- **Reports identical chains on the same line only once**
  - Ensures that duplicate chains appearing on a single line are not reported multiple times.
- **Handles different variable types in chains**
  - Verifies detection of chains that involve a mix of dictionaries, lists, tuples, and nested structures.
- **Correctly applies custom threshold values**
  - Ensures that adjusting the threshold parameter correctly affects detection behavior.
- **Verifies result structure and metadata**
  - Confirms that detected chains return correctly formatted results, including message ID, type, and occurrence details.

The test cases for this module can be found here.

4.1.4 Repeated Calls

Goal: The repeated calls detection module identifies instances where function or method calls are redundantly executed within the same scope, leading to unnecessary performance overhead. It ensures that developers receive precise recommendations for caching results when applicable. The following unit tests validate the accuracy of this functionality.

The tests assess the module's ability to detect redundant function calls, ignore functionally distinct calls, handle object state changes, and recognize cases where caching would not be beneficial. Edge cases, such as external function calls, built-in function invocations, and method calls on different objects, are also considered.

Target requirement(s): FR2, FR3, PR-PAR2, PR-PAR3 [7]

- **Detecting Repeated Function Calls**
  - Ensures the detection of repeated function calls with identical arguments within the same scope.
  - Validates that function calls on different arguments are not incorrectly flagged.
- **Detecting Repeated Method Calls**
  - Ensures that method calls on the same object instance are detected when repeated within a function.
  - Verifies that method calls on different object instances are not falsely flagged.
- **Handling Object State Modifications**
  - Ensures that function calls on objects with modified attributes between calls are not flagged.
  - Verifies that method calls are only flagged when they occur without intervening state changes.
- **Detecting External Function Calls**
  - Detects redundant external function calls such as `len(data.get("key"))`.
  - Ensures that only calls with matching parameters and return values are considered redundant.
- **Handling Built-in Functions**
  - Ensures that expensive built-in function calls (e.g., `max(data)`) are detected when redundant.
  - Verifies that lightweight built-in functions (e.g., `abs(-5)`) are ignored.
- **Ensuring Accuracy and Performance**
  - Ensures that detection does not introduce false positives by incorrectly flagging calls that differ in meaningful ways.
  - Verifies that detection performance scales efficiently with large codebases.

The test cases for this module can be found here.

4.1.5 Long Message Chain

Goal: The long message chain detection module identifies method call chains that exceed a predefined threshold (default: 5 calls). The following unit tests verify the detection capabilities.

The tests evaluate the detection of long message chains by verifying that sequences exceeding the threshold, such as five consecutive messages within a short time, are correctly identified.

They include edge cases, like chains just below the threshold or varying in length, to assess the robustness of the detection logic

Target requirement(s): FR2 [7]

- **Detects exact five calls chain**
  Ensures that a method chain with exactly five calls is flagged.
- **Detects six calls chain**
  Verifies that a chain with six method calls is detected as a smell.
- **Ignores chain of four calls**
  Ensures that a chain with only four calls (below threshold) is not flagged.
- **Detects chain with attributes and calls**
  Tests detection of a chain that involves both attribute access and method calls.
- **Detects chain inside a loop**
  Ensures detection of a chain meeting the threshold when inside a loop.
- **Detects multiple chains on one line**
  Verifies that only the first long chain on a single line is reported.
- **Ignores separate statements**
  Ensures that separate method calls across multiple statements are not mistakenly combined into a single chain.
- **Ignores short chain comprehension**
  Ensures that a short chain within a list comprehension is not flagged.
- **Detects long chain comprehension**
  Verifies that a list comprehension with a long method chain is detected.
- **Detects five separate long chains**
  Ensures that multiple long chains on separate lines within the same function are individually detected.
- **Ignores element access chains**
  Confirms that attribute and index lookups without method calls are not flagged.

The test cases for this module can be found here

4.1.6 Long Lambda Element

Goal: The following unit tests verify the correct detection of long lambda functions based on expression count and character length thresholds.

The goal of this test suite is to ensure the detection system accurately identifies long lambda functions based on predefined complexity thresholds, such as expression count and character length. It verifies that the detection logic is precise, avoiding false positives for trivial or short lambdas while correctly flagging complex or lengthy ones. The tests cover edge cases, including nested lambdas and inline lambdas passed as arguments to functions like map or filter. Additionally, the suite ensures degenerate cases, such as empty or extremely short lambdas, are not mistakenly flagged. By validating these scenarios, the detection system maintains robustness and reliability in identifying problematic lambda expressions.

Target requirement(s): FR2 [7]

- **No lambdas present**
  Ensures that when no lambda functions exist in the code, no smells are detected.
- **Short single lambda**
  Confirms that a single short lambda (well under the length threshold) with only one expression is not flagged.
- **Lambda exceeding expression count**
  Detects a lambda function that contains multiple expressions, exceeding the threshold for complexity.
- **Lambda exceeding character length**
  Identifies a lambda function that surpasses the maximum allowed character length, making it difficult to read.
- **Lambda exceeding both thresholds**
  Flags a lambda function that is both too long in character length and contains too many expressions.
- **Nested lambda functions**
  Ensures that both outer and inner nested lambdas are properly detected as long expressions.
- **Inline lambda passed to function**
  Detects lambda functions that are passed inline to functions like `map` and `filter` when they exceed the complexity thresholds.
- **Trivially short lambda function**
  Verifies that degenerate cases, such as a lambda with no real body or trivial operations, are not mistakenly flagged.

The test cases for this module can be found here

4.2 CodeCarbon Measurement Module

Goal: The CodeCarbon Measurement module is designed to measure the carbon emissions associated with running a specific piece of code. It integrates with the CodeCarbon library, which calculates the energy consumption based on the execution of a Python script.

The tests validate the functionality of the system in measuring and handling energy consumption data for code execution. They ensure that the system correctly tracks carbon emissions using the CodeCarbon library, handles both successful and failed subprocess executions, processes emissions data from CSV files, and appropriately logs all relevant events.

Target requirement(s): FR5, FR6,PR-RFT1, PR-SCR1 [7]

- **Handle CodeCarbon Measurements with a Valid File Path**
  - When a valid file path is provided, the system correctly invokes the subprocess for CodeCarbon.
  - The start and stop API endpoints of the EmissionsTracker are called, and a success message is logged.
- **Handle CodeCarbon Measurements with a Valid File Path that Causes a Subprocess Failure**
  - The subprocess is invoked even if an error occurs during the file execution.
  - The system logs an error message and the emissions data is set to None when execution fails.
- **Results Produced by CodeCarbon Run at a Valid CSV File Path and Can Be Read**
  - The emissions data can be read successfully from a valid CSV file produced by CodeCarbon.
  - The function returns the last row of emissions data.
- **Results Produced by CodeCarbon Run at a Valid CSV File Path but the File Cannot Be Read**
  - An error message is logged when the CSV file cannot be read.
  - The function returns `None` if reading the CSV file fails.
- **Given CSV Path for Results Produced by CodeCarbon Does Not Have a File**
  - When the given CSV path does not point to an existing file, an error message is logged.
  - The function returns `None` when the file is missing.

The test cases for this module can be found here.

4.3 Refactoring Module

4.3.1 Refactorer Controller

Goal: These tests verify the behavior of the RefactorerController in various scenarios, ensuring that it handles refactoring correctly and interacts with external components as expected.

Target requirement(s): FR5, UHR-UPL1 [7]

- **Verifying Successful Refactoring with a Valid Refactorer**
  - Ensures that the RefactorerController correctly runs the refactorer when a valid refactorer is available.
  - Checks that the logger captures the refactoring event.
  - Ensures the refactor method is called with the expected arguments.
  - Verifies that the modified file path is correctly generated.
- **Handling Case When No Refactorer is Found**
  - Ensures that if no refactorer is found for a given smell, the RefactorerController raises a `NotImplementedError`.
  - Confirms that the appropriate error message is logged.
- **Handling Multiple Refactorer Calls for the Same Smell**
  - Tests the behavior when the refactorer is called multiple times for the same smell.
  - Ensures that the smell counter is updated correctly.
  - Verifies that unique file names are generated for each call.
- **Verifying the Behavior When Overwrite is Disabled**
  - Ensures that when the overwrite flag is set to false, the refactor method is called with the correct argument.
  - Verifies that files are not overwritten when this option is disabled.
- **Handling Case Where No Files are Modified**
  - Checks that when no files are modified by the refactorer, the RefactorerController correctly returns an empty list of modified files.

The test cases for this module can be found here.

4.3.2 String Concatenation in a Loop

Goal: The refactoring module transforms inefficient string concatenation into a more performant approach using list accumulation and `''.join()`. Unit tests for this module ensure that:

Target requirement(s): FR3, FR6 [7]

- **Proper Initialization of the List**
  - The string variable being concatenated is replaced with a list at the start of the loop.
  - The list is initialized to an empty list if the initial string is obviously empty.
  - Otherwise, the list is initialized with the initial string as the first item.
- **Correct Usage of List Methods in the Loop**
  - The `append()` method is used instead of assignments inside the loop.
  - The `insert()` method is used to insert values at the beginning of the string.
  - If the string is re-initialized inside the loop, the `clear()` method is used to empty the list before re-initializing it.
  - If multiple concatenations happen in the same loop, each is accumulated in the correct list.
- **Correct String Joining After the Loop**
  - The accumulated list is joined using `''.join(list)` after the loop.
  - The final assignment replaces the original concatenation variable with the joined string.
- **Handling of Edge Cases from Detection**
  - If the concatenation variable is accessed inside the loop, no refactoring is applied.
  - Conditional concatenations result in conditional list appends.
  - Order-sensitive concatenations (e.g., prefix insertions) preserve the original behavior.
- **Preserving Readability and Maintainability**
  - The refactored code maintains readability and does not introduce unnecessary complexity.
  - Nested loop modifications preserve the correct scoping of the refactored lists.
  - The refactored code maintains proper formatting and indentation.
  - Unnecessary modifications to unrelated code are avoided.

The test cases for this module can be found here.

4.3.3 Long Element Chain

Goal: The long element chain refactoring module simplifies deeply nested dictionary accesses by flattening them into top-level keys while preserving functionality. The following unit tests verify the correctness of this transformation.

The tests assess the ability to detect and refactor long element chains by verifying correct dictionary transformations, accessing pattern updates, and handling of various data structures. Edge cases, such as shallow accesses, multiple affected files, and mixed depths of access, are included to ensure robustness.

Target requirement(s): FR3, FR6, PR-PAR3 [7]

- **Identifies and Refactors Basic Nested Dictionary Access**
  - Ensures that deeply nested dictionary keys are detected and refactored correctly.
- **Refactors Dictionary Accesses Across Multiple Files**
  - Verifies that dictionary changes propagate correctly when a deeply nested dictionary is accessed in different files.
- **Handles Dictionary Access via Class Attributes**
  - Ensures that nested dictionary accesses within class attributes are correctly detected and refactored.
- **Ignores Shallow Dictionary Accesses**
  - Verifies that dictionary accesses below the predefined threshold remain unchanged.
- **Handles Multiple Long Element Chains in the Same File**
  - Ensures that all occurrences of excessive dictionary accesses in a file are refactored individually.
- **Detects and Refactors Mixed Access Depths**
  - Confirms that the module correctly differentiates and processes deep accesses while ignoring shallow ones.
- **Validates Resulting Metadata and Formatting**
  - Confirms that the refactored output includes well-structured results, such as correct message IDs, types, and occurrences.

The test cases for this module can be found here.

4.3.4 Member Ignoring Method

Goal: The member ignoring method refactoring module ensures that methods that do not reference instance attributes or methods are converted into static methods. This transformation improves code clarity, enforces proper design patterns, and eliminates unnecessary instance bindings. The following unit tests validate the correctness of this functionality.

The tests assess the correct detection of methods that can be converted to static methods, proper removal of `self` as a parameter, correct updating of method calls, and handling of inheritance. Edge cases such as instance-dependent methods, overridden methods, and preserving existing decorators are also considered.

Target requirement(s): FR3, FR6 [7]

- **Correct Addition of the `@staticmethod` Decorator**
  - The method receives the `@staticmethod` decorator when it does not use `self` or any instance attributes.
  - The decorator is added directly above the method definition, preserving any existing decorators.
- **Removal of the `self` Parameter**
  - The first parameter of the method is removed if it is named `self`.
  - Other parameters remain unchanged.
  - The method signature is correctly adjusted to reflect the removal.
- **Modify Instance Calls to the Method**
  - Instance objects of the class calling the method will be modified to use a static call directly from the class itself.
  - Calls in a different file from the one where the smell was detected will also have the appropriate calls modified.
- **Handling of Methods in Classes with Inheritance**
  - If a subclass instance calls the method, the call is also modified.
  - If a method is overridden in a subclass, refactoring is **not** applied.
- **Ensuring No Modification to Instance-Dependent Methods**
  - Methods that reference `self` directly (e.g., `self.attr`, `self.method()`) are not modified.
  - Methods that indirectly access instance attributes (e.g., through another method call) are left unchanged.

- **Preserving Readability and Maintainability**
  - The refactored code maintains proper formatting and indentation.
  - Unnecessary modifications to unrelated code are avoided.

The test cases for this module can be found here.

4.3.5 Use a Generator

Goal: The use-a-generator refactoring module optimizes list comprehensions used within functions like `all()` and `any()` by transforming them into generator expressions. This refactoring improves memory efficiency while maintaining code correctness. The following unit tests validate the accuracy of this functionality.

The tests ensure that list comprehensions inside `all()` and `any()` calls are correctly converted to generator expressions while preserving original behavior. Additional test cases verify proper handling of multiline comprehensions, nested conditions, and various iterable types. Edge cases, such as improperly formatted comprehensions or comprehensions spanning multiple lines, are also considered.

Target requirement(s): FR3, FR5, FR6, PR-PAR3 [7]

- **Refactoring List Comprehensions Inside `all()` Calls**
  - Ensures that list comprehensions within `all()` are transformed into generator expressions.
  - Validates that the transformation preserves original functionality.
- **Refactoring List Comprehensions Inside `any()` Calls**
  - Ensures that list comprehensions within `any()` are transformed into generator expressions.
  - Verifies that the modified code remains functionally identical to the original.
- **Handling Multi-line List Comprehensions**
  - Ensures that multi-line list comprehensions are refactored correctly.
  - Verifies that proper indentation and formatting are preserved after transformation.
- **Handling Edge Cases**
  - Ensures that improperly formatted comprehensions do not result in errors.
  - Confirms that refactoring does not introduce unnecessary modifications or change unrelated code.

- **Ensuring Correct Formatting and Readability**
  - Validates that refactored code adheres to Python's style guidelines.
  - Ensures that readability and maintainability are preserved after refactoring.

The test cases for this module can be found here.

4.3.6 Cache Repeated Calls

Goal: The cache repeated calls refactoring module optimizes redundant function and method calls by storing results in local variables, reducing unnecessary recomputation. This refactoring enhances performance by eliminating duplicate executions of expensive operations. The following unit tests validate the accuracy of this functionality.

The tests ensure that function and method calls that produce identical results are cached and replaced with stored values where applicable. Additional test cases verify proper handling of method calls on objects, preserving object state, and integrating with function arguments. Edge cases, such as function calls with varying inputs, method calls on different instances, and presence of docstrings, are also considered.

Target requirement(s): FR3, FR5, FR6, PR-PAR3, PR-PAR2 [7]

- **Refactoring Repeated Function Calls**
  - Ensures that repeated function calls within a scope are replaced with cached results.
  - Validates that caching is only applied when function arguments remain unchanged.
- **Refactoring Repeated Method Calls**
  - Ensures that method calls on the same object instance are cached and replaced with stored values.
  - Verifies that method calls on different object instances are not incorrectly refactored.
- **Handling Object State Modifications**
  - Ensures that method calls on objects whose attributes change between calls are not cached.
  - Verifies that method calls are only cached when no state changes occur.
- **Handling Edge Cases**
  - Ensures that function calls with varying arguments are not incorrectly cached.
  - Confirms that caching does not interfere with function scope, closures, or nested function calls.

- **Refactoring in the Presence of Docstrings**
  - Ensures that refactoring does not alter function behavior when docstrings are present.
  - Verifies that caching maintains readability and proper formatting.
- **Ensuring Correct Formatting and Readability**
  - Validates that refactored code adheres to Python's style guidelines.
  - Ensures that readability and maintainability are preserved after refactoring.

The test cases for this module can be found here.

4.3.7 Long Parameter List

Goal: The Long Parameter List refactoring module replaces function definitions and calls involving a large number of parameters with grouped parameter instances. This enhances the efficiency of the code as related parameters are grouped together into instantiated parameters. The validity of the functionality can be ensured through unit tests.

The tests ensure that all cases for function declarations, including constructors, instance methods, static methods and standalone functions are updated with the group parameter instances. Similarly, corresponding calls to these functions as well as references to original parameters are preserved. The refactored result also preserves use of default values in function signature and positional arguments in function calls.x

Target requirement(s): FR3, FR6 [7]

- **Handle Constructor with More Parameters than Configured Limit, All Used**
  - The refactor correctly handles constructors with 8 parameters, including a mix of positional and keyword arguments.
  - Function signature and instantiation are updated to include all 8 parameters.
- **Handle Constructor More Parameters than Configured Limit, some of which are Unused**
  - Constructor correctly handles unused parameters by excluding them from the updated signature and instantiation.
  - The refactor updates the constructor and function body to properly reflect the used parameters.
- **Handle Instance Method More Parameters than Configured Limit (2 Defaults)**

  - The refactor correctly handles instance methods with 8 parameters, including default values for some.
  - The method signature and instantiation are updated to preserve the default values while reflecting the used parameters.

- **Handle Instance Method Refactor with More Parameters than Configured Limit, some of which are Unused**
  - Unused parameters in the instance method are correctly excluded and grouped into new data and config parameter classes.
  - Function maintains the correct structure and that the method now accepts the new class objects instead of individual parameters.

- **Handle Static Method Refactor with More Parameters than Configured Limit, some of which are Unused and some have Default Values**
  - Unused parameters are correctly excluded from the static method signature and instantiation.
  - Parameters with default values are properly maintained in the updated method signature and body.

- **Handle Standalone Function Refactor with More Parameters than Configured Limit, some of which are Unused**
  - Unused parameter is excluded from the function signature after the refactor.
  - Refactored function uses grouped parameter classes to handle the remaining parameters efficiently.

The test cases for this module can be found here.

4.3.8 Long Message Chain

Goal: The following unit tests verify the correctness of refactoring long element chains into more readable and efficient code while maintaining valid Python syntax.

This test suite focuses on validating the refactoring of long method chains into intermediate variables, improving code readability while preserving functionality. It ensures that simple method chains are split correctly and that special cases, such as f-strings or chains with arguments, are handled appropriately. The tests also verify that refactoring maintains proper indentation, especially within nested blocks like if statements or loops. Additionally, the suite confirms that refactoring does not alter the behavior of the original code, even in contexts like print statements. By testing both long and short chains, the suite ensures consistency and correctness across various scenarios.

Target requirement(s): FR5, FR6, FR3 [7]

- **Basic method chain refactoring**
  Ensures that a simple method chain is refactored correctly into separate intermediate variables.

- **F-string chain refactoring**
  Verifies that method chains applied to f-strings are properly broken down while preserving correctness.

- **Modifications even if the chain is not long**
  Ensures that method chains are refactored consistently, even if they do not exceed the length threshold.

- **Proper indentation preserved**
  Confirms that the refactored code maintains the correct indentation when inside a block statement such as an `if` condition.

- **Method chain with arguments**
  Tests that method chains containing arguments (e.g., `replace("H", "J")`) are correctly refactored.

- **Print statement preservation**
  Ensures that method chains within a `print` statement are refactored without altering their functionality.

- **Nested method chains**
  Verifies that nested method chains (e.g., method calls on method results) are properly refactored into intermediate variables.

The test cases for this module can be found here

4.3.9 Long Lambda Element

Goal: The following unit tests verify the correctness of refactoring long lambda expressions into named functions while maintaining valid Python syntax and preserving code functionality.

The goal of this test suite is to ensure long lambda expressions are refactored into named functions while maintaining code functionality, readability, and proper syntax. It verifies that simple single-line lambdas are converted correctly and that more complex cases, such as multi-line lambdas or those with multiple parameters, are handled appropriately. The tests ensure that refactoring preserves the original behavior of the code, even for lambdas used as keyword arguments or passed to functions like map or reduce. Additionally, the suite confirms that no unnecessary changes, such as added print statements, are introduced during refactoring. By covering a wide range of cases, the suite ensures the refactoring process is both reliable and effective.

Target requirement(s): FR5, FR6, FR3 [7]

- **Basic lambda conversion**
  Verifies that a simple single-line lambda expression is correctly converted into a named function with proper indentation and structure.
- **No extra print statements**
  Ensures that the refactoring process does not introduce unnecessary print statements when converting lambda expressions.
- **Lambda in function argument**
  Tests that lambda expressions used as arguments to other functions (e.g., in `map()` calls) are properly refactored while maintaining the original function call structure.
- **Multi-argument lambda**
  Verifies that lambda expressions with multiple parameters are correctly converted into named functions with the appropriate parameter list.
- **Lambda with keyword arguments**
  Ensures that lambda expressions used as keyword arguments in function calls are properly refactored while preserving the original keyword argument syntax and indentation.
- **Very long lambda function**
  Tests the refactoring of complex, multi-line lambda expressions with extensive mathematical operations, verifying that the converted function maintains the original logic and structure.

The test cases for this module can be found here

4.4 VsCode Plugin

4.4.1 Detect Smells Command

Goal: The detect smells command is responsible for initiating the smell detection process, retrieving results from the backend, and ensuring proper highlighting in the VS Code editor. The command must handle various scenarios, including caching, missing editor instances, and server failures, while providing meaningful feedback to the user. The following unit tests verify its accuracy.

The tests assess the correct fetching and caching of smells, proper interaction with the highlighting module, handling of missing files or inactive editors, and the system's ability to recover from server downtime. Edge cases such as rapidly changing file hashes and updates to enabled smells are also considered.

Target requirement(s): FR10, OER-IAS1 [7]

- **Handling of Missing Active Editor**
  - The command shows an error message when no active editor is found.

- **Handling of Missing File Path**
  - The command shows an error message when the active editor has no valid file path.
- **Handling of No Enabled Smells**
  - The command shows a warning message when no smells are enabled in the configuration.
- **Using Cached Smells**
  - The command uses cached smells when the file hash and enabled smells match the cached data.
  - The command highlights the cached smells in the editor.
- **Fetching New Smells**
  - The command fetches new smells when the file hash changes or enabled smells are updated.
  - The command updates the cache with the new smells and highlights them in the editor.
- **Handling of Server Downtime**
  - The command shows a warning message when the server is down and no cached smells are available.
- **Highlighting Smells**
  - The command highlights detected smells in the editor when smells are found.
  - The command shows a success message with the number of highlighted smells.

The test cases for this module can be found here.

4.4.2 Refactor Smells Command

Goal: The refactorSmell command is responsible for refactoring code areas identified as ”smells” in a project. It works by refactoring areas in code that could benefit from refactoring (smells) that are chosen by the user. The process involves multiple steps, including saving the file, calling a backend refactoring service to refactor the identified smell, updating any relevant data, and initiating a refactor preview to the user.

The tests assess the correct fetching and caching of smells, proper interaction with the highlighting module, handling of missing files or inactive editors, and the system’s ability to recover from server downtime. Edge cases such as rapidly changing file hashes and updates to enabled smells are also considered.

Target requirement(s): FR5, FR6, FR10, PR-RFT1, PR-RFT2 [7]

- **No Active Editor Found**
  - The command correctly handles the case where there is no active editor open.
  - An appropriate error message is shown when no editor or file path is available.
- **Attempting to Refactor When No Smells Are Detected**
  - The command does not proceed when no smells are detected in the file.
  - An error message is shown indicating that no smells are detected for refactoring.
- **Attempting to Refactor When Selected Line Doesn't Match Any Smell**
  - The command doesn't proceed if the selected line doesn't match any detected smell.
  - An error message is shown to inform the user that no matching smell was found for refactoring.
- **Refactoring a Smell When Found on the Selected Line**
  - The command successfully saves the current file and triggers the refactoring of a detected smell.
  - The `refactorSmell` method is called with the correct parameters, and the refactored preview is shown to the user.
- **Handling API Failure During Refactoring**
  - The command gracefully handles API failures during the refactoring process.
  - An error message is displayed to the user if refactoring fails, with the appropriate details logged.

The test cases for this module can be found here.

4.4.3 Document Hashing

Goal: The document hashing module is responsible for generating and managing document hashes to track changes in files. By ensuring efficient tracking, this module allows caching mechanisms to work correctly and prevents unnecessary reprocessing. The following unit tests validate its correctness.

The tests verify the module's ability to detect file modifications, handle new files, and prevent redundant updates when no changes occur. Edge cases such as rapid sequential edits, hash mismatches, and multiple concurrent document updates are also considered.

Target requirement(s): FR10, OER-IAS1 [7]

- **Handling of Unchanged Document Hashes**
  - The module does not update the workspace storage if the document hash has not changed.
  - The existing hash is retained for the document.
- **Handling of Changed Document Hashes**
  - The module updates the workspace storage when the document hash changes.
  - The new hash is correctly calculated and stored.
- **Handling of New Documents**
  - The module updates the workspace storage when no hash exists for the document.
  - A new hash is generated and stored for the document.

The test cases for this module can be found here.

4.4.4 File Highlighter

Goal: The file highlighter module enhances code visibility by applying visual decorations to highlight detected code smells in the editor. It ensures that identified issues are clearly distinguishable while preserving readability. The following unit tests verify the correctness of this functionality.

The tests assess the correct creation of decorations, accurate application of highlighting based on detected smells, proper handling of initial and subsequent highlights, and the removal of outdated decorations. Edge cases such as overlapping decorations and incorrect style applications are also considered.

Target requirement(s): FR10, OER-IAS1, LFR-AP2 [7]

- **Creation of Decorations**
  - Decorations are created with the correct color and style for each type of code smell.
  - The decoration type is properly initialized and can be applied to the editor.
- **Highlighting Smells**
  - Smells are highlighted in the editor based on their line occurrences.
  - The hover content for each smell is correctly associated with the decoration.
- **Handling of Initial Highlighting**
  - On the first initialization, decorations are applied without resetting existing ones.

  - The decorations are properly stored for future updates.

- **Resetting Decorations**
  - On subsequent calls, existing decorations are disposed of before applying new ones.
  - The reset process ensures no overlapping or redundant decorations.

The test cases for this module can be found here.

4.4.5 Hover Manager

Goal: The Hover Manager module manages hover functionality for Python files, providing hover content for detected code smells and allowing refactoring commands. The following unit tests verify the accuracy of this functionality.

The tests assess the ability of the `HoverManager` to register hover providers, handle hover content, update smells, generate refactor commands, and ensure correct hover content formatting. Edge cases, such as no smells and the correct propagation of updates to smells, are also considered.

Target requirement(s): LFR-AP2 [7]

- **Registers hover provider for Python files**
  - Verifies that the `HoverManager` correctly registers a hover provider for Python files.
- **Subscribes hover provider correctly**
  - Ensures that the hover provider is correctly subscribed to the context manager.
- **Returns null for hover content if there are no smells**
  - Checks that `getHoverContent` returns `null` when no smells are detected for a file.
- **Updates smells when `getInstance` is called again**
  - Verifies that when `getInstance` is called again with new smells, the manager updates the stored smells and returns the same instance.
- **Updates smells correctly**
  - Confirms that `updateSmells` correctly updates the list of smells in the manager.
- **Generates valid hover content**

  - Ensures that `getHoverContent` generates valid hover content with correctly formatted smell details, including refactor commands and proper structure.

- **Registers refactor commands**
  - Verifies that the refactor commands are properly registered for individual and all smells of a specific type.

The test cases for this module can be found here.

4.4.6 Line Selection Manager

Goal: The Line Selection Manager module provides functionality for detecting and commenting on code smells based on a line selection. The following unit tests verify the correctness of this functionality.

The tests assess the ability of the `LineSelectionManager` to handle various scenarios such as missing editor instances, multiple smells on a line, single-line vs. multi-line selections, and correct comment generation. Edge cases, such as mismatched document hashes, non-existent smells, and the absence of selected text, are also considered.

Target requirement(s): UHR-EOU1 [7]

- **Removes last comment if decoration exists**
  - Verifies that the last comment decoration is removed correctly if one exists, ensuring that the decoration is disposed of properly.
- **Does not proceed if no editor is provided**
  - Ensures that no action is taken when `commentLine` is called with `null` as the editor.
- **Does not add comment if no smells detected for file**
  - Checks that no comment is added when no smells are detected for the current file, confirming that no unnecessary decorations are applied.
- **Does not add comment if document hash does not match**
  - Verifies that no comment is added when the document hash in the workspace data does not match the hash of the document, ensuring that the editor's state remains consistent with the expected data.
- **Does not add comment for multi-line selections**
  - Tests that no comment is added when there is a multi-line selection, ensuring that only single-line selections are processed.

- **Does not add comment when no smells exist at line**
  - Ensures that no comment is added when no smells are associated with the selected line, preventing unnecessary decorations from being applied.
- **Displays single smell comment without count**
  - Verifies that a single smell is displayed correctly without a count, confirming that the decoration is applied with the correct content format.
- **Adds a single-line comment if a smell is found**
  - Confirms that a single-line comment is added correctly when a smell is found on the selected line, ensuring proper decoration application.
- **Displays a combined comment if multiple smells exist**
  - Verifies that a combined comment is displayed when multiple smells exist on the same line.
  - Ensures that the decoration is created with the correct formatting and applied to the correct range.

The test cases for this module can be found here.

4.4.7 Handle Smells Settings

Goal: The VS Code settings management module enables users to customize detection settings, update enabled smells, and ensure workspace consistency. This module integrates with the IDE to provide real-time updates when settings change. The following unit tests validate the correctness of this functionality.

The tests ensure that enabled smells are correctly retrieved from user configurations, updates to settings trigger the appropriate notifications, and changes are accurately reflected in the workspace. Additional test cases verify proper handling of missing configurations, format conversions, and cache clearing operations. Edge cases, such as unchanged settings and invalid inputs, are also considered.

Target requirement(s): FR10, UHR-PSI1, UHR-EOU2, OER-IAS1 [7]

- **Retrieving Enabled Smells**
  - Ensures that the function retrieves all enabled smells from the user's VS Code settings.
  - Validates that the function returns an empty object when no smells are enabled.
- **Handling Updates to Smell Filters**

  - Ensures that enabling a smell triggers a notification to the user.
  - Confirms that disabling a smell results in a proper update message.
  - Validates that when no changes occur, no unnecessary cache updates are performed.

- **Clearing Cache on Settings Update**
  - Ensures that enabling or disabling a smell triggers a workspace cache wipe.
  - Confirms that cache clearing only occurs when actual changes are made.

- **Formatting Smell Names**
  - Ensures that smell names stored in kebab-case are correctly formatted into a readable format.
  - Verifies that empty input results in an empty string without errors.

- **Ensuring User-Friendly Notifications**
  - Confirms that updates to smell settings provide clear, informative messages.
  - Ensures that error handling follows polite and constructive messaging principles.

The test cases for this module can be found here.

4.4.8 Wipe Workspace Cache

Goal: The ”Wipe Workspace Cache” command is responsible for clearing specific caches related to the workspace in a development environment, primarily to reset the state of stored data such as ”smells” and file changes. It can be triggered for different reasons, which determine which caches are cleared and how the command behaves. It also updates file hashes for visible editors when appropriate. Upon successful execution, a corresponding success message is shown to the user. In case of an error, an error message is displayed to the user.

The tests ensure that appropriate caches are being cleared as and when instructed.

Target requirement(s): FR3, FR5, FR8 [7]

- **Wipe Cache with No Reason Provided**
  - Only the ”smells” cache is cleared when no reason is provided.
  - A success message indicating the workspace cache has been successfully wiped is displayed.

- **Wipe Cache with Reason ”manual”**
  - Both the ”smells” and ”file changes” caches are cleared when the reason is ”manual.”

  - A success message indicating the workspace cache was manually wiped is shown.

- **Wipe Cache When No Files Are Open**
  - The command correctly handles the case when there are no open files.
  - A log message is generated indicating that no open files are available to update.

- **Wipe Cache with Open Files**
  - When there are open files, a message indicating the number of visible files is logged.
  - The hashes for each open file are updated as expected.

- **Wipe Cache with Reason "settings"**
  - Only the "smells" cache is cleared when the reason is "settings."
  - A success message is shown indicating the cache was wiped due to changes in smell detection settings.

- **Wipe Cache When an Error Occurs**
  - An error message is logged when an error occurs during the cache wipe process.
  - a user-facing error message is displayed, indicating the failure to wipe the workspace cache.

The test cases for this module can be found here.

4.4.9 Backend

Goal: The backend handles interactions with the backend server for tasks such as checking server status, initializing logs, fetching detected smells, and refactoring those smells in the code.

The tests ensure that the system correctly interacts with the backend to check the server's status, initialize logging, fetch code smells, and perform refactoring tasks. These tests confirm that the server status is accurately updated based on successful or failed responses, that log initialization behaves as expected under different conditions, and that the system correctly handles and processes detected smells for refactoring.

Target requirement(s): FR6, PR-SCR1, PR-RFT1, PR-RFT2 [7]

- **Handle Server Status Check with Successful Response**
  - When the server responds successfully, the status is set to `ServerStatusType.UP`.
  - The correct update of server status to indicate that the server is operational.

- **Handle Server Status Check with Error or Failure**
  - When the server responds with an error or fails to respond, the status is set to `ServerStatusType.DOWN`.
  - Correctly handles a failed server response by ensuring the status reflects the server's downtime.
- **Handle Initiation of Logs with Valid Directory Path (Success)**
  - When a valid directory path is provided and the backend responds successfully, the function returns `true`, indicating successful log initialization.
  - System can successfully initialize logs with the backend and sync them.
- **Handle Initiation of Logs with Valid Directory Path (Failure)**
  - When the backend fails to initialize logs, the function returns false.
  - Properly handles to provide feedback if the log initialization process fails.

The test cases for this module can be found here.

4.5 API Routes

4.5.1 Smell Detection Endpoint

Goal: The smell detection endpoint provides an API for retrieving detected code smells from the backend. It ensures efficient communication with the smell detection module while handling errors gracefully. The following unit tests verify the accuracy of this functionality.

The tests assess the correctness of the endpoint's response structure, error handling for missing files, validation of request data, and handling of internal exceptions. Edge cases, such as malformed requests and empty responses, are also considered.

Target requirement(s): FR10, OER-IAS1 [7]

- **Successful Detection of Smells**
  - The API endpoint returns a successful response when the file exists and smells are detected.
  - The response contains the correct number of detected smells and adheres to the expected data structure.
- **Handling of File Not Found Errors**
  - The API endpoint returns an appropriate error response when the specified file does not exist.
  - The error message clearly indicates that the file was not found.

- **Handling of Internal Server Errors**
  - The API endpoint returns an error response when an unexpected exception occurs during smell detection.
  - The error message indicates an internal server error.
- **Validation of Input Data**
  - The API validates the presence and correctness of required fields in the request.
  - The API rejects invalid input with appropriate error messages.

The test cases for this module can be found here.

4.5.2 Refactoring Endpoint

Goal: The refactoring endpoint provides an API for refactoring detected code smells by leveraging the backend refactoring module. It ensures that refactored code is returned efficiently while verifying energy savings. The following unit tests validate the accuracy of this functionality.

The tests assess the correctness of refactored output, proper retrieval of energy measurements, error handling for missing source directories, and graceful failures when unexpected conditions arise. Edge cases such as unsuccessful refactorings and unchanged energy consumption are also considered.

Target requirement(s): FR10, OER-IAS1 [7]

- **Successful Refactoring Process**
  - The API endpoint returns a successful response when the refactoring process completes without errors.
  - The response includes the refactored data and updated smells.
  - Energy measurements are correctly retrieved and compared to ensure energy savings.
- **Handling of Source Directory Not Found**
  - The API endpoint returns an appropriate error response when the specified source directory does not exist.
  - The error message clearly indicates that the directory was not found.
- **Handling of Energy Measurement Failures**
  - The API endpoint returns an error response when initial or final energy measurements cannot be retrieved.

  - The API endpoint returns an error response when no energy savings are detected after refactoring.

- **Handling of Unexpected Errors**
  - The API endpoint returns an error response when an unexpected exception occurs during the refactoring process.
  - The error message includes details about the exception.

The test cases for this module can be found here.

References

[1] Astral. Ruff. https://docs.astral.sh/ruff/. Accessed: 2025-02-18.

[2] CircleCI. Using pytest with circleci. https://circleci.com/blog/pytest-python-testing/. Accessed: 2024-11-03.

[3] DataCamp. Memory profiling with python. https://www.datacamp.com/tutorial/memory-profiling-python. Accessed: 2024-11-03.

[4] M. S. et al. pytest-cov documentation. https://pytest-cov.readthedocs.io/en/latest/. Accessed: 2025-02-18.

[5] N. Kuruparan, S. Walker, M. Hussain, A. Amin, and T. Brar. Module guide. 2024. URL https://github.com/ssm-lab/capstone--source-code-optimizer/blob/main/docs/Design/SoftArchitecture/MG.pdf.

[6] N. Kuruparan, S. Walker, M. Hussain, A. Amin, and T. Brar. Module interface specification. 2024. URL https://github.com/ssm-lab/capstone--source-code-optimizer/blob/main/docs/Design/SoftDetailedDes/MIS.pdf.

[7] N. Kuruparan, S. Walker, M. Hussain, A. Amin, and T. Brar. Software requirements specification for software engineering: An eco-friendly source code optimizer. 2024. URL https://github.com/ssm-lab/capstone--source-code-optimizer/blob/main/docs/SRS/SRS.pdf.

[8] Microsoft. Testing extensions. https://code.visualstudio.com/api/working-with-extensions/testing-extension. Accessed: 2025-02-24.

[9] OpenJs Foundation. Eslint - find and fix problems in your javascript code. https://eslint.org/. Accessed: 2025-02-18.

[10] PrettierCode. Prettier - opinionated code formatter. https://prettier.io/. Accessed: 2025-02-18.

[11] Web Accessibility Initiative, EOWG, and AGWG. Web content accessibility guidelines (wcag). Technical report, Web Accessibility Initiative (WAI), 2024. URL https://www.w3.org/WAI/standards-guidelines/wcag/.

Appendices

A Appendix

A.1 Symbolic Parameters

Not applicable at the moment.

A.2 Usability Survey Questions

See the surveys folder under `docs/Extras/UsabilityTesting`.

B Reflection

The information in this section will be used to evaluate the team members on the graduate attribute of Lifelong Learning.

The purpose of reflection questions is to give you a chance to assess your own learning and that of your group as a whole, and to find ways to improve in the future. Reflection is an important part of the learning process. Reflection is also an essential component of a successful software development process.

Reflections are most interesting and useful when they're honest, even if the stories they tell are imperfect. You will be marked based on your depth of thought and analysis, and not based on the content of the reflections themselves. Thus, for full marks we encourage you to answer openly and honestly and to avoid simply writing "what you think the evaluator wants to hear."

Please answer the following questions. Some questions can be answered on the team level, but where appropriate, each team member should write their own response:

1. What went well while writing this deliverable?
2. What pain points did you experience during this deliverable, and how did you resolve them?
3. What knowledge and skills will the team collectively need to acquire to successfully complete the verification and validation of your project? Examples of possible knowledge and skills include dynamic testing knowledge, static testing knowledge, specific tool usage, Valgrind etc. You should look to identify at least one item for each team member.
4. For each of the knowledge areas and skills identified in the previous question, what are at least two approaches to acquiring the knowledge or mastering the skill? Of the identified approaches, which will each team member pursue, and why did they make this choice?

Mya Hussain

- *What went well while writing this deliverable?*

  Writing functional tests for the capstone project went surprisingly smoothly. I found that having a clear understanding of the project's requirements and functionalities made it easier to structure my tests logically. The existing documentation provided a solid foundation, allowing me to focus on creating relevant scenarios without needing extensive revisions.Overall, I felt a sense of accomplishment as I was able to write robust tests that will contribute to the project's success.

- *What pain points did you experience during this deliverable, and how did you resolve them?*

One challenge I faced was ensuring that each test was precise and effectively communicated its purpose. At times, I found myself overthinking the wording or structure, which slowed me down. To tackle this, I started breaking down each test into simple components, focusing on the core functionality rather than getting lost in the details. I also struggled with organizing the tests logically to create a seamless flow in the documentation. I resolved this by grouping tests thematically, which made it easier to follow. Despite the frustrations, I learned to embrace the process and appreciate the importance of thorough documentation in building a robust project. Balancing this deliverable and the POC was also challenging as there wasnt much turnaroud time between the two and we hadn't coded anything previously.

Sevhena Walker

- *What went well while writing this deliverable?*

  I was responsible for writing system tests for the projects non-functional requirements and I found the process to be very useful for gaining a deep understanding of all the qualities a system should have. When you write a requirement, obviously, there is some thought put into it, but actually writing out the test really sheds light on all the facets that go into that requirement. I feel like I have even more to contribute to my team after this deliverable.

- *What pain points did you experience during this deliverable, and how did you resolve them?*

  Writing out all those tests was extremely long, and I found myself re-writing tests more than once while pondering on the best way to test the requirements. Some tests needed to be combined due to a near identical testing process and some needed more depth. I also sometimes struggled with determining if some testing could even be feasibly done with our team's resources. To resolve this I held a discussion or 2 with my team so that we could have more brains working on the matter and to ensure that I wasn't making some important decisions unilaterally.

Nivetha Kuruparan

- *What went well while writing this deliverable?*

  Working on the Verification and Validation (VnV) plan for the Source Code Optimizer project was a pretty smooth experience overall. I really enjoyed defining the roles in section 3.1, which helped clarify what everyone was responsible for. This not only made the team feel more involved but also kept us on track with our testing strategies.

  Diving into the Design Verification Plan in section 3.3 was another highlight for me. It helped me get a better grasp of the requirements from the SRS. I felt more confident knowing we had a solid verification approach that covered all the bases, including functional and non-functional requirements. The discussions about incorporating static verification techniques and the importance of regular peer reviews were eye-opening and really enhanced our strategy for maintaining code quality.

- *What pain points did you experience during this deliverable, and how did you resolve them?*

  I struggled a bit with figuring out how to effectively integrate feedback mechanisms into our VnV plan. It was tough to think through how to keep the feedback loop going throughout development. I tackled this by setting up a clear process for documenting feedback during our code reviews and testing phases, which I included in section 3.4. This not only improved our documentation but also helped us stay committed to continuously improving as we moved forward.

Ayushi Amin

- *What went well while writing this deliverable?*

  Writing this deliverable was a really crucial part of the process. It helped me see the bigger picture of how we're going to ensure everything in the SRS gets tested properly. What went well was the clarity that came from laying out the plan step by step. Even though we haven't put it into action yet, just knowing we have a solid structure in place gives me confidence.

  Another highlight was sharing our completed sections with Dr. Istvan. It was great to get his feedback and know that he appreciated the level of detail we included. Having that validation made me feel like we're on the right track. It also reminded me how important it is to be thorough from the start, so we're not scrambling later when we're deep into testing. Having all the requiremnts and test cases mapped out helps me stress less as I know have an idea of what the proejct will look like and have these documents top guide the process in case we get stuck or forget something.

- *What pain points did you experience during this deliverable, and how did you resolve them?*

  One of the challenges was trying to anticipate potential gaps or issues in our testing process while still being in the planning phase. Thinking through how to cover both functional and non-functional requirements in the SRS in a comprehensive yet practical way was tricky. We resolved this by deciding to create a traceability matrix, which will help us ensure that every requirement is accounted for once we move into the testing phase. Even though the matrix isn't done yet, just planning to use it gives a sense of structure.

  Another tough spot was figuring out how to handle usability and performance testing in a way that doesn't feel overly theoretical. Since we're not at the implementation stage, it's hard to gauge what users will really need. To work through this, I focused on drawing from what we know about our end-users and aligning our plan with the goals outlined in the SRS. Keeping that user-centered perspective helped ground the plan, making it feel more actionable even at this early stage.

Tanveer Brar

- *What went well while writing this deliverable?*

  Clearly pointing out the tools to use for various aspects of Automated Validation and Testing(such as unit test framework, linter) has created a well-defined plan for this verification. Now the project has a structured approach to validation. Knowing the tools before implementation will allow both code quality enforcement and the gathering of coverage metrics. For the Software Validation Plan, external data source(open source Python code bases for testing) has added confidence that the validation approach would align closely with real world scenarios.

- *What pain points did you experience during this deliverable, and how did you resolve them?*

  One of the challenges was ensuring compatibility between different tools for automated testing and validation plan. For example, code coverage tool needs to be supported by the unit testing framework. To resolve this, I conducted research on all validation tools, to choose the ones that fit into the project's needs while being compatible with each other.

Group Reflection

- *What knowledge and skills will the team collectively need to acquire to successfully complete the verification and validation of your project? Examples of possible knowledge and skills include dynamic testing knowledge, static testing knowledge, specific tool usage, Valgrind etc. You should look to identify at least one item for each team member.*

  Sevhena will need to deepen her understanding of test coordination and project tracking using GitHub Issues. She'll focus on creating detailed issue templates for various testing stages, managing the workflow through Kanban boards, and using labels and milestones effectively to track progress. Additionally, mastering test case documentation and ensuring efficient communication through GitHub's discussion and comment features will be critical.

  Mya will enhance her skills in functional testing by learning to write comprehensive test cases directly linked to GitHub Issues. She will leverage GitHub Actions to automate repetitive functional tests and integrate them into the development workflow. Familiarity with continuous integration pipelines and how they relate to functional testing will help her verify that all functional requirements are met consistently.

Ayushi will focus on integration testing by ensuring that the Python package, VSCode plugin, and GitHub Action work together seamlessly. She'll develop expertise in using PyJoules to assess energy efficiency during integration tests and learn to create automated workflows via GitHub Actions. Ensuring smooth integration of PyTorch models and maintaining consistent coding standards with Pylint will be essential. She'll also manage dependencies and coordinate with the team using GitHub's multi-repository capabilities.

Tanveer will deepen her knowledge of performance testing using PyJoules to monitor and optimize energy consumption. She will also need to develop skills in security testing, ensuring that the Python code adheres to best security practices, possibly integrating tools like Bandit along with Pylint for static code analysis. Setting up and maintaining performance benchmarks using GitHub Issues will ensure transparency and continuous improvement.

Nivetha will enhance her skills in usability and user experience testing, particularly in evaluating the intuitiveness of the VSCode plugin interface. She will focus on collecting and analyzing user feedback, linking it to GitHub Issues to drive interface improvements. Documenting user experience testing and ensuring that the product's UI meets user expectations will be a significant part of her role. Using Pylint to maintain consistent code quality in user-facing components will also be essential.

Istvan will provide oversight by monitoring the team's progress, using GitHub Insights to ensure that testing processes meet industry standards. He will guide the team in integrating PyJoules, Pylint, and PyTorch effectively into the V&V workflow, offering feedback and ensuring alignment with project goals.

All group members will have to learn how to use pytests to perform test cases in this entire project.

- *For each of the knowledge areas and skills identified in the previous question, what are at least two approaches to acquiring the knowledge or mastering the skill? Of the identified approaches, which will each team member pursue, and why did they make this choice?*

  Sevhena Walker (Lead Tester)

  - **Knowledge Areas:** Test coordination, PyJoules, GitHub Actions, Pylint.
  - **Approaches:**
    - Online Courses and Tutorials: Enroll in courses focused on test automation, PyJoules, and GitHub Actions.
    - Hands-on Practice: Apply knowledge directly by setting up test cases and automation workflows in the project.
  - **Preferred Approach:** Hands-on Practice
  - **Reason:** This approach allows her to see immediate results and iterate quickly, building confidence in her coordination and automation skills.

Mya Hussain (Functional Requirements Tester)

- **Knowledge Areas:** PyTorch, functional testing, GitHub Actions, Pylint.
- **Approaches:**
  - Technical Documentation and Community Forums: Study PyTorch documentation and participate in forums like Stack Overflow.
  - Mentorship and Collaboration: Pair with experienced team members or mentors to get guidance and feedback on functional testing practices.
- **Preferred Approach:** Technical Documentation and Community Forums
- **Reason:** It allows her to explore topics deeply and find solutions to specific issues, promoting self-sufficiency.

Ayushi Amin (Integration Tester)

- **Knowledge Areas:** PyJoules, integration testing, PyTorch, GitHub Actions.
- **Approaches:**
  - Workshops and Webinars: Attend live or recorded sessions focused on energy-efficient software development and integration testing techniques.
  - Project-Based Learning: Directly work on integrating components and iteratively improving based on project needs.
- **Preferred Approach:** Project-Based Learning
- **Reason:** It aligns with her role's focus on real-world integration, providing relevant experience and immediate feedback.

Tanveer Brar (Non-Functional Requirements Tester - Performance/Security)

- **Knowledge Areas:** Performance testing with PyJoules, security testing, Pylint.
- **Approaches:**
  - Specialized Training Programs: Join programs or bootcamps that focus on performance and security testing.
  - Peer Learning: Collaborate with team members and participate in knowledge-sharing sessions.
- **Preferred Approach:** Peer Learning
- **Reason:** It promotes team synergy and allows him to gain practical insights from those working on similar tasks.

Nivetha Kuruparan (Non-Functional Requirements Tester - Usability/UI)

- **Knowledge Areas:** Usability testing, user experience, GitHub Issues, Pylint.
- **Approaches:**
  - User Feedback Analysis: Conduct regular user testing sessions and analyze feedback.

    * Online UX/UI Design Courses: Enroll in courses that focus on usability principles and user experience design.

  - **Preferred Approach:** User Feedback Analysis
  - **Reason:** This approach provides real-world insights into how the product is perceived and used, making adjustments more relevant.

Istvan David (Supervisor)

  - **Knowledge Areas:** Supervising V&V processes, providing feedback, ensuring industry standards.
  - **Approaches:**
    * Industry Conferences and Seminars: Attend events focused on software verification and validation trends.
    * Continuous Professional Development: Engage in regular self-study and professional development activities.
  - **Preferred Approach:** Continuous Professional Development
  - **Reason:** This method allows for a consistent update of skills and knowledge aligned with evolving industry standards.