System Verification and Validation Plan for Software Engineering

Team 4, EcoOptimizers

Nivetha Kuruparan
Sevhena Walker
Tanveer Brar
Mya Hussain
Ayushi Amin

November 4, 2024

Revision History

Date	Version	Notes
November 4th, 2024	0.0	Created initial revision of VnV Plan

Contents

List of Tables

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` for writing and executing these tests.

- **Static Code Analysis**: To maintain high code quality, static analysis tools such as `Pylint` and `Flake8` 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(`coverage.py` 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 [2].

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` will be used, as it is easy to install and includes built-in support for visual display of output [3].

Static Analyzer: The codebase will be statically analyzed using the PyLint tool, as it is easy to integrate with most IDEs and is actively maintained (as opposed to PySmells). PyLint provides a wide range of support, including error detection, refactoring suggestions, and code style enforcement, making it a strong choice for static analysis [8].

Code Coverage Tools and Plan for Summary: The code base will be analyzed to determine the percentage of code executed during tests. For granular-level coverage, `coverage.py` will be used, as it supports branch, line, and path coverage. Additionally, `coverage.py` is a test framework-independent, allowing integration with the project's unit test framework, Pytest.
Initially the aim is to achieve a 40% coverage and gradually increment the level with time. Weekly reports generated from `coverage.py` will be used to track coverage trends and set goals accordingly to address any gaps in testing in the growing codebase.

Test Coverage Tools: The project will use `TestRail`, a test case management tool, to provide traceability from test cases to requirements, ensuring all requirements are covered by tests. Additionally, TestRail can help run tests and track results in integration with Pytest [4].

Linters: To enforce the official Python PEP 8 style guide, the team will use `PyLint`, which is also the choice for static analysis of the code.

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` as well as `PyLint` checks on every code push. Through automated testing, any errors and code smells 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: Automatic
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: Automatic
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: Automatic
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.

test-FR-IA-4 Test for Original Code Passing the Original Test Suite

Control: Automatic
Initial State: Idle.
Input: Python code and its associated test suite.
Output: The original code passes 100% of the test suite.

Test Case Derivation: This test ensures that the original code is functional and compliant with the provided test suite, confirming that the input code is valid.

How test will be performed:

(a) The original code will be executed against its associated test suite.

(b) Verify that all tests in the original test suite pass, indicating that the original code is valid and functioning as expected.

test-FR-IA-5 Valid Python Test Suite Acceptance

Control: Automatic
Initial State: Tool is idle.
Input: A valid Python test suite file (`testSuite.py`) with valid syntax and tests.
Output: The system accepts the test suite and confirms it is ready for execution.

Test Case Derivation: Confirms that the tool can accept a valid test suite as input, as required by FR 2.

How test will be performed: Load a valid test suite `.py` file into the tool and observe that it is accepted without errors.

test-FR-IA-6 Feedback for Test Suite with Invalid Syntax

Control: Automatic
Initial State: Tool is idle.
Input: A test suite file (`invalid_test_suite.py`) containing syntax errors.
Output: The system rejects the test suite and provides an error message detailing the syntax issue.

Test Case Derivation: Verifies the tool's capability to identify and report errors in test suites, meeting FR 2.

How test will be performed: Load a test suite file with syntax errors into the tool and check for appropriate error reporting.

test-FR-IA-7 Test Suite with No Test Cases

Control: Automatic
Initial State: Tool is idle.
Input: A valid Python file (`empty_test_suite.py`) that contains no test cases.
Output: The system rejects the file and provides an error message indicating that there are

no test cases present.

Test Case Derivation: Ensures the tool identifies test suites lacking test cases, complying with FR 2.

How test will be performed: Load a test suite file with no defined test cases and verify that the system produces an appropriate error message.

3.1.2 Code Smell Detection Tests

This area includes tests to verify the detection of specified code smells that impact energy efficiency (FR 2).

test-FR-CSD-1 Detection of Large Class (LC)

Control: Automatic
Initial State: Tool has loaded a `.py` file containing a class with many methods and attributes.
Input: Python file with a class that exceeds the threshold for ”Large Class” smell.
Output: Tool identifies the ”Large Class” smell and suggests refactoring options like breaking the class into smaller classes.

Test Case Derivation: Ensures ”Large Class” code smells are identified and appropriately refactored.

How test will be performed: Load a file with a large class and verify detection.

test-FR-CSD-2 Detection of Long Parameter List (LPL)

Control: Automatic
Initial State: Tool has loaded a `.py` file containing a method with a long parameter list.
Input: Python file with a method using more parameters than the threshold.
Output: Tool flags the ”Long Parameter List” smell and suggests bundling parameters into objects or reducing parameters.

Test Case Derivation: Ensures ”Long Parameter List” code smell detection.

How test will be performed: Load a file with a method having a long parameter list and confirm detection.

test-FR-CSD-3 Detection of Long Method (LM)

Control: Automatic
Initial State: Tool has loaded a `.py` file with a method that exceeds the line limit threshold.
Input: Python file containing a long method.
Output: Tool detects ”Long Method” and suggests breaking it into smaller methods.

Test Case Derivation: Ensures ”Long Method” detection and suggestions for improving readability.

How test will be performed: Load a file with a long method and check for detection.

test-FR-CSD-4 Detection of Long Message Chain (LMC)

Control: Automatic
Initial State: Tool has loaded a `.py` file with a chain of method calls.
Input: Python file containing a message chain exceeding the threshold.
Output: Tool flags the "Long Message Chain" smell and suggests ways to simplify it, such as introducing intermediary methods.

Test Case Derivation: Validates "Long Message Chain" detection and suggestions for code simplification.

How test will be performed: Load a file with a long chain of method calls and confirm detection.

test-FR-CSD-5 Detection of Long Scope Chaining (LSC)

Control: Automatic
Initial State: Tool has loaded a `.py` file containing deeply nested scopes.
Input: Python file with excessive scope chaining.
Output: Tool detects "Long Scope Chaining" and suggests reducing nesting or refactoring.

Test Case Derivation: Ensures tool detects deep nesting and provides ways to make code more readable.

How test will be performed: Load a file with nested scopes and confirm detection.

test-FR-CSD-6 Detection of Long Base Class List (LBCL)

Control: Automatic
Initial State: Tool has loaded a `.py` file with a class that inherits from many base classes.
Input: Python file containing a class with an extensive inheritance list.
Output: Tool flags "Long Base Class List" and suggests refactoring, such as restructuring inheritance.

Test Case Derivation: Validates that long inheritance lists are detected, and refactoring options are provided.

How test will be performed: Load a file with a long base class list and confirm detection.

test-FR-CSD-7 Detection of Useless Exception Handling (UEH)

Control: Automatic
Initial State: Tool has loaded a `.py` file with empty or redundant `try-except` blocks.
Input: Python file containing useless exception handling blocks.
Output: Tool flags "Useless Exception Handling" and suggests meaningful handling or removal.

Test Case Derivation: Confirms detection of redundant exception handling and refactoring options.

How test will be performed: Load a file with empty `try-except` blocks and verify detection.

test-FR-CSD-8 Detection of Long Lambda Function (LLF)

Control: Automatic
Initial State: Tool has loaded a `.py` file containing lambda functions that exceed the line or complexity threshold.
Input: Python file with a long lambda function.
Output: Tool detects "Long Lambda Function" and suggests converting it to a named function.

Test Case Derivation: Validates detection of long lambda functions and refactoring suggestions for clarity.

How test will be performed: Load a file with a long lambda and verify detection.

test-FR-CSD-9 Detection of Complex List Comprehension (CLC)

Control: Automatic
Initial State: Tool has loaded a `.py` file with list comprehensions containing nested conditions.
Input: Python file with a complex list comprehension.
Output: Tool flags "Complex List Comprehension" and suggests simplifying the expression.

Test Case Derivation: Ensures tool detects complex list comprehensions and suggests simplifications.

How test will be performed: Load a file with complex list comprehension and confirm detection.

test-FR-CSD-10 Detection of Long Element Chain (LEC)

Control: Automatic
Initial State: Tool has loaded a `.py` file with a long sequence of chained elements (e.g., dictionary access).
Input: Python file containing an element chain exceeding the length threshold.
Output: Tool detects "Long Element Chain" and suggests restructuring the code for readability.

Test Case Derivation: Confirms tool detects long element chains and suggests simplification.

How test will be performed: Load a file with a long element chain and verify detection.

test-FR-CSD-11 Detection of Long Ternary Conditional Expression (LTCE)

Control: Automatic
Initial State: Tool has loaded a `.py` file with a ternary conditional expression that exceeds the line or complexity threshold.
Input: Python file containing a long ternary conditional.
Output: Tool flags "Long Ternary Conditional Expression" and suggests converting to a standard `if-else` block.

Test Case Derivation: Ensures long ternary expressions are detected, and refactoring options are provided.

How test will be performed: Load a file with a long ternary expression and confirm detection.

test-FR-CSD-12 No Code Smells Detected Handling

Control: Automatic
Initial State: Tool is idle.
Input: A valid Python file (filename.py) that adheres to best practices and contains no detectable code smells.
Output: The system returns a message indicating that no code smells were found in the code.

Test Case Derivation: This test ensures that the tool can correctly identify when there are no code smells present, as per functional requirement FR 2.

How test will be performed: Provide a Python file that is well-structured and free of common code smells, and verify that the tool outputs a message confirming the absence of smells.

3.1.3 Refactoring Suggestion (RS) Tests

The following tests aim to validate the tool's capability to suggest appropriate refactorings in response to identified code smells, as outlined in the functional requirements (FR 5). These tests ensure that for each detected code smell, the tool provides actionable code modifications that not only enhance maintainability but also lead to measurable reductions in energy consumption.

test-FR-RS-1 Large Class (LC) RS

Control: Automatic
Initial State: Tool has identified a large class in the provided Python file.
Input: A Python file containing a class with a high number of lines of code and methods.
Output: The tool suggests splitting the large class into smaller, more manageable classes and displays the suggested modifications.

Test Case Derivation: Ensures that the tool provides a refactoring suggestion that reduces energy consumption while maintaining functionality as per FR 5.

How test will be performed: Feed a Python file with a large class to the tool and verify that it displays a refactoring suggestion to break the class into smaller parts.

test-FR-RS-2 Long Parameter List (LPL) RS

Control: Automatic
Initial State: Tool has detected a method with too many parameters.

Input: A Python file with a method signature that contains an excessive number of parameters.
Output: The tool suggests using a data structure (e.g., a dictionary or an object) to encapsulate the parameters and shows the modified method signature.

Test Case Derivation: Confirms that the tool can identify long parameter lists and suggest refactoring to improve code clarity and energy efficiency.

How test will be performed: Submit a Python file with a method featuring a long parameter list and check that the tool provides a refactoring suggestion.

test-FR-RS-3 Long Method (LM) RS

Control: Automatic
Initial State: Tool has identified a method that exceeds a predefined line count.
Input: A Python file containing a method that is excessively long.
Output: The tool suggests breaking the long method into smaller methods and displays the proposed modifications.

Test Case Derivation: Validates that the tool recognizes long methods and suggests refactoring to enhance maintainability and reduce energy usage.

How test will be performed: Provide the tool with a Python file containing a long method and observe if it suggests breaking it into smaller methods with clear modifications.

test-FR-RS-4 Long Message Chain (LMC) RS

Control: Automatic
Initial State: Tool has identified a long message chain in the code.
Input: A Python file with chained method calls resulting in a long message chain.
Output: The tool suggests simplifying the message chain by assigning intermediate results to variables and shows the refactored code.

Test Case Derivation: Ensures that the tool can detect long message chains and provide effective refactoring suggestions that improve energy efficiency.

How test will be performed: Use a Python file containing a long message chain and confirm that the tool displays a suggestion to simplify it.

test-FR-RS-5 Long Scope Chaining (LSC) RS

Control: Automatic
Initial State: Tool has detected a long scope chain in the provided code.
Input: A Python file with multiple nested function calls or scope references.
Output: The tool suggests flattening the scope chain by refactoring into clearer, standalone function calls and displays the proposed modifications.

Test Case Derivation: Confirms that the tool can identify long scope chaining and suggest refactoring to enhance clarity and maintainability.

How test will be performed: Feed a Python file with a long scope chain to the tool and check if it suggests appropriate refactoring.

test-FR-RS-6 Long Base Class List (LBCL) RS

Control: Automatic
Initial State: Tool has identified a class inheriting from many base classes.
Input: A Python file with a class declaration that inherits from multiple base classes.
Output: The tool suggests refactoring the class to reduce the number of base classes, possibly by using composition instead of inheritance, and displays the suggested changes.

Test Case Derivation: Validates that the tool recognizes long base class lists and provides suggestions for improving class design and energy efficiency.

How test will be performed: Provide the tool with a Python file containing a class with a long base class list and observe if it suggests refactoring.

test-FR-RS-7 Useless Exception Handling (UEH) RS

Control: Automatic
Initial State: Tool has detected unnecessary exception handling in the code.
Input: A Python file containing try-except blocks that do not provide meaningful handling.
Output: The tool suggests removing or modifying the exception handling and displays the modified code.

Test Case Derivation: Validates that the tool can identify useless exception handling and suggests actionable refactorings to enhance clarity and efficiency.

How test will be performed: Feed the tool a Python file with unnecessary exception handling and check if it suggests modifications.

test-FR-RS-8 Long Lambda Function (LLF) RS

Control: Automatic
Initial State: Tool has identified a lambda function that is too complex or lengthy.
Input: A Python file containing a long or complex lambda function.
Output: The tool suggests refactoring the lambda function into a named function and shows the proposed changes.

Test Case Derivation: Confirms that the tool recognizes long lambda functions and provides suggestions for refactoring to enhance readability and performance.

How test will be performed: Submit a Python file with a long lambda function to the tool and verify that it suggests converting it into a named function.

test-FR-RS-9 Complex List Comprehension (CLC) RS

Control: Automatic
Initial State: Tool has detected a complex list comprehension.
Input: A Python file containing a list comprehension that is hard to read or understand.
Output: The tool suggests breaking the list comprehension into a for loop and displays the modified code.

Test Case Derivation: Ensures that the tool identifies complex list comprehensions and suggests refactoring for clarity and efficiency.

How test will be performed: Provide a Python file with a complex list comprehension and observe if the tool offers a simpler alternative.

test-FR-RS-10 Long Element Chain (LEC) RS

Control: Automatic
Initial State: Tool has detected a long chain of method calls on an object.
Input: A Python file with an object undergoing multiple chained calls.
Output: The tool suggests breaking the chain into separate calls and displays the refactored code.

Test Case Derivation: Validates that the tool can recognize long element chains and provide suggestions to improve code structure and efficiency.

How test will be performed: Use a Python file featuring a long element chain and verify that the tool suggests breaking it apart.

test-FR-RS-11 Long Ternary Conditional Expression (LTCE) RS

Control: Automatic
Initial State: Tool has detected a complex ternary conditional expression.
Input: A Python file with a long ternary expression that is difficult to read.
Output: The tool suggests refactoring the ternary expression into a standard if-else statement and shows the suggested changes.

Test Case Derivation: Ensures that the tool identifies long ternary conditional expressions and provides clearer refactoring alternatives.

How test will be performed: Submit a Python file containing a long ternary expression to the tool and confirm that it suggests refactoring it into an if-else statement.

test-FR-RS-12 Energy Consumption Measurement for Suggested Refactoring

Control: Automatic
Initial State: Tool has suggested a refactoring for detected code smells.
Input: Suggested refactored code.
Output: Measurement showing improved energy consumption in joules.

Test Case Derivation: Confirms suggestions provide measurable energy efficiency improvement, per FR 5.

How test will be performed: Apply a refactoring and measure energy consumption before and after.

test-FR-RS-13 Optimal Refactoring Selection

Control: Automatic
Initial State: Tool has identified multiple refactoring options for a single code smell.
Input: Multiple refactoring options.
Output: System chooses the refactoring with the lowest measured energy consumption.

Test Case Derivation: Ensures that the tool optimizes for energy efficiency, meeting FR 7.

How test will be performed: Provide multiple refactoring options and verify the tool selects the most energy-efficient one.

test-FR-RS-14 RS Not Possible Handling

Control: Automatic
Initial State: Tool is idle.
Input: Code containing a complex smell that cannot be refactored due to constraints.
Output: The system provides a message indicating that no refactoring suggestions can be made for the identified smell or given code.

Test Case Derivation: Ensures the tool gracefully handles situations where refactoring is too complex or not feasible.

How test will be performed: Provide a code example that includes a complex smell and observe the output for an appropriate message regarding the lack of suggestions.

test-FR-RS-15 Selection of Identical Energy Consumption Refactorings

Control: Automatic
Initial State: The refactoring tool has analyzed a Python code file and identified multiple minimal refactorings.
Input: Two or more refactorings that result in the same minimal energy consumption improvement.
Output: The tool randomly selects one refactoring to apply.

Test Case Derivation: This test ensures that when multiple refactorings provide the same energy efficiency gain, the tool correctly implements one of them without preference, thereby fulfilling FR 5.

How test will be performed: The tool will be run on a Python file containing code smells. It will be observed whether it selects one of the refactorings with identical energy improvements for application.

3.1.4 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 Validate Refactored Code Functionality On Provided Test Suite

Control: Automatic
Initial State: The original Python code is equipped with an existing test suite it passes.
Input: The original Python code and its associated test suite.
Output: The refactored code passes 100% of the original test suite.

Test Case Derivation: This test confirms that the refactored code preserves the original functionality by passing all tests from the original suite, as stipulated in FR 3.

How test will be performed: The tool will refactor the code, and then the original test suite will be executed against the refactored code to check for passing results.

test-FR-OV-2 Verification of Valid Python Output

Control: Automatic
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.

3.1.5 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.

3.1.6 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.

3.1.7 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 Colour-coded refactoring indicators for energy savings

Type: Non-Functional, Manual, Dynamic
Initial State: IDE plugin open with sample code loaded
Input/Condition: The plugin displays refactoring suggestions based on energy savings
Output/Result: Refactoring suggestions are colour-coded according to energy-saving potential (e.g., yellow for minor savings, red for major savings)

How test will be performed: The tester will load sample code with multiple refactoring suggestions based on energy-saving potential and activate the plugin's analysis feature. The tester will then review each suggestion, confirming that they are visually differentiated by colour codes (e.g., yellow for minor savings and red for major savings). They will interact with each coloured indicator to ensure that it is responsive and accurately represents the suggested energy savings levels.

test-LF-4 Visual alerts in GitHub Action for significant energy savings

Type: Non-Functional, Manual, Dynamic
Initial State: GitHub Action enabled for a pull request (PR)
Input/Condition: PR analysis indicates energy savings exceeding a predefined threshold.
Output/Result: A success icon or green label appears in the PR summary

How test will be performed: The tester will set up a pull request (PR) with changes that yield significant energy savings. When the GitHub Action completes the analysis, the tester will check the PR summary to confirm that a green label or success icon appears, indicating substantial energy savings. The tester will repeat the test with a PR that does not meet the threshold to ensure no alert is displayed.

test-LF-5 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)[9].

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 Audio cues for important actions

Type: Non-Functional, Manual, Dynamic
Initial State: IDE plugin open with audio cues enabled
Input/Condition: User performs actions triggering audio cues
Output/Result: The system emits an audible attention catching sound.

How test will be performed: The tester will enable audio cues in the tool's settings, then perform a series of tasks, such as running code analysis, applying refactorings, and saving changes. Each action should trigger an audio cue indicating task completion or user feedback. The tester will evaluate the volume, timing, and appropriateness of each cue and document whether the cues enhance the user experience or cause any distractions.

test-UH-6 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-7 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-8 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-9 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 projects of varying sizes ready (1,000, 5,000, 10,000, 100,000 lines of code).
Input/Condition: Initiate the refactoring process for each project sequentially
Output/Result: Process completes within 15 seconds for projects up to 5,000 lines of code, 20 seconds for 10,000 lines of code and within 2 minutes for 100,000 lines of code.

How test will be performed: The tester will use four python projects of different sizes: small (1,000 lines), medium (5,000 and 10,000 lines), and large (100,000 lines). For each project, 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 each project is checked against the expected result.

test-PF-2 Integrity of refactored code against runtime errors

Type: Non-Functional, Automated, Dynamic
Initial State: Refactoring tool ready, with user-provided code and test suite loaded
Input/Condition: User initiates refactoring on the input code
Output/Result: Refactored code passes all tests in the user-provided suite without runtime errors and adheres to Python syntax standards

How test will be performed: The refactoring tool will first apply the refactoring to the user-provided code. After refactoring, an automated test suite will run, confirming that all original tests pass, indicating no loss of functionality. The refactored code will then be validated by an automatic linter to ensure compliance with Python syntax standards.

test-PF-3 Functionality preservation post-refactoring

Type: Non-Functional, Automated, Dynamic
Initial State: Python file ready for refactoring with proper configurations for the system
Input/Condition: User initiates refactoring on the code file
Output/Result: The refactored code should pass 100% of user-provided tests

How test will be performed: see test test-FR-OV-1

test-PF-4 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-5 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

test-PF-6 Handling unexpected inputs

Type: Non-Functional, Manual, Dynamic
Initial State: IDE open and ready with various non-standard and invalid input files
Input/Condition: User attempts to refactor invalid code files and non-Python files
Output/Result: Tool detects invalid input, displays a clear error message, and does not crash

How test will be performed: The tester will sequentially give any of the following invalid files as input to the system :

- Non-Python files (e.g., .txt, .java, .cpp, .js)
- Invalid Python files with syntax errors (e.g., unmatched brackets, improper indentation)
- Corrupted files that contain random symbols or partially deleted code

For each file type, the tester will initiate the refactoring process and observe the tool's response. The tool should detect each invalid input, display an error message describing the issue, and recover from the error without crashing.

test-PF-7 Fallback Options for Failed Refactoring Attempts

Type: Non-Functional, Manual, Dynamic
Initial State: The tool is set up in an IDE with a sample code file that includes code smells
Input/Condition: User initiates a refactoring process on the sample code file
Output/Result: The tool logs failed refactoring attempts, provides a clear error notification, and suggests alternative refactoring options without interrupting the overall process.

How test will be performed: The tester will load a sample code file into the tool that contains code smells. Upon initiating the refactoring, the tester will observe the tool's response to any failed attempts, verifying that it logs the error. The tool should then attempt alternative refactorings without restarting the process. The tester will document the clarity of the error message, the relevance of alternative suggestions, and confirm that the tool remains functional, supporting uninterrupted refactoring of other code smells.

test-PF-8 Maintainability and Adaptability of the Tool

Objective: Ensure that the tool's codebase is structured to support future updates for new Python versions and evolving coding standards, minimizing the effort required for maintenance.

Scope: This test applies to the tool's code structure, documentation quality, and modularity to facilitate adaptability and maintainability over time.

Methodology: Code walkthrough and static analysis

Process:

- Conduct a code walkthrough to evaluate the modular structure of the codebase, verifying that components are organized to allow independent updates.
- Review code comments, documentation, and naming conventions to ensure clarity and consistency, supporting ease of understanding for future developers.

- Identify any dependencies on specific Python versions and assess the ease of updating these components for compatibility with newer versions.
- Document any gaps in modularity or documentation and consult with the development team on improvements to support maintainability.

Roles and Responsibilities: The development team will conduct the code review and documentation assessment, with the project supervisor overseeing and validating improvements for long-term adaptability.

Tools and Resources: Code editor, documentation templates, Python development guidelines, and coding standards

Acceptance Criteria: The codebase is modular, well-documented, and adaptable, allowing for straightforward updates with minimal impact on existing functionality.

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 Emissions Standards Compliance

Objective: Ensure that the tool's emissions metrics and reports align with widely used standards (e.g., GRI 305, GHG, ISO 14064) to support users in environmental compliance and sustainability tracking.

Scope: This test applies to the tool's metrics and reporting components, including data format and labelling in the emissions report.

Methodology: Static analysis and documentation walkthrough

Process:

- Review emissions metrics in the tool's documentation and compare them with requirements from GRI 305, GHG, and ISO 14064 standards.
- Verify that all required emissions metrics from these standards are present in the tool's reports, with proper format and units.
- Confirm that all emissions categories and labels align with standard definitions to ensure consistency and accuracy.

Roles and Responsibilities: The development team and project supervisor will conduct the documentation review and patch any discrepancies.

Tools and Resources: Tool's user guide, sample emissions reports, GRI 305, GHG, and ISO 14064 standards documentation

Acceptance Criteria: The tool's emissions metrics meet or exceed the coverage required by GRI 305, GHG, and ISO 14064 standards. All labels and units are accurate, consistent, and aligned with these standards.

test-OPE-2 Integration with GitHub Actions for automated refactoring

Type: Non-Functional, Automated, Dynamic
Initial State: GitHub repository with access to the refactoring library in GitHub Actions
Input/Condition: User sets up a GitHub Actions workflow that calls the refactoring library
Output/Result: GitHub Actions successfully initiates refactoring processes through the library as part of a continuous integration workflow

How test will be performed: The tester will configure a GitHub Actions workflow in a test repository, specifying steps to call the refactoring library. After committing a sample code change, the workflow should trigger automatically. The tester will verify that the refactoring library runs within GitHub Actions, completes the refactoring process, and provides feedback in the workflow logs. Successful integration will be confirmed by viewing refactoring results directly within the GitHub Actions logs.

test-OPE-3 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-4 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-5 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 User authentication before accessing tool features

Type: Non-Functional, Manual, Dynamic
Initial State: System installed, user unauthenticated
Input/Condition: User attempts to submit code or view refactoring reports
Output/Result: Access is denied

How test will be performed: The tester will first attempt to submit code and access refactored reports without logging in, verifying that access is denied. The tester will then log in using valid company credentials and repeat the actions to confirm access is granted only after successful authentication.

test-SRT-2 Internal-Only Communication with Energy and Reinforcement Learning Tools

Objective: Ensure that the refactoring tool communicates exclusively with the internal energy consumption tool and reinforcement learning model, without exposing any public API endpoints.

Scope: This test applies to all network and API interactions between the refactoring tool and internal services, ensuring no direct access is available to users or external applications.

Methodology: Code walkthrough and static analysis

Process:

- Conduct a code walkthrough of the network and API components, focusing on the access control configurations for the energy consumption tool and reinforcement learning model.
- Inspect the code for any exposed API endpoints or network configurations that might allow external access.
- Attempt to access the internal tools directly from an external environment, ensuring that all external attempts are blocked.
- Verify that the tool's communication is contained within internal environments and restricted to authorized system components.

Roles and Responsibilities: The development team will conduct the code review and testing, ensuring secure access protocols.

Tools and Resources: Access to the codebase, network configuration files, and security audit tools

Acceptance Criteria: No public or external API endpoints exist for the internal tools, and only the refactoring tool can access the energy consumption and reinforcement learning models.

test-SRT-3 Preventing Unauthorized Changes to Refactored Code and Reports

Objective: Ensure the tool's refactored code and energy reports are protected from any unauthorized external modifications, maintaining data integrity and user trust.

Scope: This test applies to the data security of refactored code and energy report storage layers, verifying that access is restricted to authorized users and processes only.

Methodology: Static analysis and code walkthrough

Process:

- Review the codebase and database configurations to verify the implementation of access controls and data security measures.
- Confirm that the tool's security settings prevent any unauthorized external modifications, maintaining data integrity across all storage layers.
- Document any vulnerabilities found and evaluate with the development team to ensure improvements are made where necessary.

Roles and Responsibilities: The development team will conduct the code review while the project supervisor will oversee the test results and approve any necessary security enhancements.

Tools and Resources: Access to security configuration files, code editor

Acceptance Criteria: The review attendees find no egregious faults within the system that might allow unauthorized external access or modifications to refactored code and energy report data.

test-SRT-4 Notification and consent for data handling

Type: Non-Functional, Manual, Dynamic
Initial State: System idle
Input/Condition: User initiates refactoring on their source code
Output/Result: Tool displays data handling notice and requests explicit consent before data collection

How test will be performed: The tester will begin the refactoring process, and the tool should present a notice explaining data collection, storage, and processing practices, in compliance with PIPEDA. The user must provide explicit consent before proceeding. The tester will confirm that no data collection occurs until consent is granted.

test-SRT-5 Confidential Handling of User Data in Compliance with PIPEDA

Objective: Ensure that all user-submitted data, energy reports, and refactored code are treated as confidential, encrypted during storage and transmission, and managed according to PIPEDA.

Scope: This test applies to the tool's data handling practices, specifically the encryption protocols for transmission and storage, and data modification options for user compliance requests.

Methodology: Code walkthrough and static analysis

Process:

- Review the encryption settings in the codebase to confirm that all data related to user submissions, energy reports, and refactored code is encrypted during transmission and storage.
- Verify that an option is available for users to request modifications to their personal data as per PIPEDA requirements.
- Document any gaps in data security or user request handling, and collaborate with the development team to implement improvements as needed.

Roles and Responsibilities: The development team will conduct the code review and implement any necessary improvements, with the project supervisor overseeing the compliance with PIPEDA standards.

Tools and Resources: Access to encryption libraries, security configuration files

Acceptance Criteria: All user data is encrypted during storage and transmission, and users have a reliable method for requesting data modifications as per PIPEDA specifications.

test-SRT-6 Audit Logs for User Actions

Objective: Ensure the tool maintains tamper-proof logs of key user actions, including code submissions, login events, and access to refactored code and reports, to ensure accountability and traceability.

Scope: This test applies to the logging mechanisms for user actions, focusing on the security and tamper-proof nature of logs.

Methodology: Code walkthrough and static analysis

Process:

- Review the logging mechanisms within the codebase to confirm that events such as logins, code submissions, and report accesses are properly recorded with timestamps and user identifiers.
- Document the integrity of the logs and any vulnerabilities found, and collaborate with the development team on any necessary improvements.

Roles and Responsibilities: The development team will conduct the code review, with oversight by the project supervisor to verify that logging mechanisms meet security requirements.

Tools and Resources: Access to log files, logging library documentation, security testing tools

Acceptance Criteria: Logs are tamper-proof, recording all critical user actions with integrity, and resistant to unauthorized modifications.

test-SRT-7 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.

test-SRT-8 Immunity Against Malware and Unauthorized Programs

Type: Non-Functional, Automated, Dynamic
Initial State: The tool is deployed in a controlled test environment with security protocols enabled, ready to be tested against simulated malware attacks.
Input/Condition: Simulated malware attacks are executed using Atomic Red Team by Red Canary[1], targeting vulnerabilities such as unauthorized data access, process interference, and data tampering.
Output/Result: The tool detects, blocks, and logs all simulated malware activities without any compromise to data integrity or tool functionality.

How test will be performed: The tester will deploy the tool in a secure, isolated test environment and initiate simulated malware attacks using Atomic Red Team. Each simulation will mimic various malware behaviours, including attempts to access or modify data and disrupt the refactoring process. The tester will observe and document the tool's responses to each simulated attack, verifying that it blocks unauthorized actions, maintains data integrity, and logs the events for traceability.

3.2.7 Cultural

The following subsection tests cover all Cultural requirements listed in the SRS [7]. These test are to ensure that the tool is accessible and appropriate for a global audience, avoiding any culturally sensitive or inappropriate elements.

test-CULT-1 Cultural sensitivity of icons and colours

Type: Non-Functional, Manual, Dynamic
Initial State: IDE plugin open
Input/Condition: User interacts with the plugin
Output/Result: More than 60% of users give an answer greater than 3 the survey question targeting this test.

How test will be performed: Users complete tasks requiring prompts and answer the survey found in A.2 on cultural sensitivity of the interface design.

test-CULT-2 Support for metric and imperial units

Type: Non-Functional, Manual, Dynamic
Initial State: Tool ready with energy consumption metrics displayed in the default unit system
Input/Condition: User toggles the measurement units between metric and imperial
Output/Result: Energy consumption measurements update correctly between metric and imperial units

How test will be performed: The tester will navigate to the settings, locate the measurement unit toggle, and switch between metric and imperial units. After each toggle, the displayed energy consumption data should reflect the correct measurement units. The tester will validate accuracy by comparing values against known conversions to ensure the toggle functions accurately and smoothly.

test-CULT-3 Cultural sensitivity of content

Type: Non-Functional, Manual, Dynamic
Initial State: IDE plugin open
Input/Condition: User interacts with the plugin
Output/Result: More than 60% of users give an answer greater than 3 the survey question targeting this test.

How test will be performed: Users complete tasks requiring prompts and answer the survey found in A.2 on cultural sensitivity of the content of the system.

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 collection, usage, storage, and communication practices are fully compliant with the Personal Information Protection and Electronic Documents Act (PIPEDA) and Canada's Anti-Spam Legislation (CASL), to avoid legal penalties and enhance user trust.

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

Methodology: Documentation walkthrough and static analysis

Process:

- Review the tool's data handling and storage protocols to confirm compliance with PIPEDA, particularly focusing on secure storage, data usage transparency, and privacy rights.
- Verify the presence of a user consent mechanism that informs users of data collection and provides options for managing their data.
- Inspect communication practices to ensure compliance with CASL, confirming that the tool provides users with notification and opt-in options for all communications.
- Document any gaps in compliance and consult with the development team for required adjustments.

Roles and Responsibilities: The development team will conduct the compliance review and implement any necessary updates.

Tools and Resources: Access to documentation on PIPEDA and CASL requirements, tool's data handling and communication protocols, test user accounts for opt-in verification

Acceptance Criteria: The tool complies with all PIPEDA and CASL requirements, with secure data handling, user consent options, and compliant communication practices.

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

Objective: Ensure the tool's quality management and software development processes align with ISO 9001 for quality management and SSADM (Structured Systems Analysis and Design Method) standards for software development, building stakeholder trust and market acceptance.

Scope: This test covers the tool's adherence to ISO 9001 quality management practices and SSADM methodologies for software development processes.

Methodology: Documentation walkthrough and code walkthrough

Process:

- Conduct a review of the tool's quality management procedures to verify alignment with ISO 9001 standards, including documentation, testing, and feedback mechanisms.
- Examine software development workflows to confirm adherence to SSADM standards, focusing on design, analysis, and structured development practices.
- Identify any deviations from ISO 9001 and SSADM requirements, document these findings, and discuss necessary adjustments with the development team.
- Validate improvements in quality management and software development after implementing recommendations.

Roles and Responsibilities: The development team will conduct the standards compliance review, and the project supervisor will oversee the review process.

Tools and Resources: Access to ISO 9001 and SSADM standards documentation, project quality management records, and development workflows

Acceptance Criteria: The tool's quality management and software development processes fully adhere to ISO 9001 and SSADM standards, supporting a high-quality, structured approach to development.

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

This will be done after the Detailed Design.

References

[1] Canary. Atomic red team, 2023. URL https://github.com/redcanaryco/atomic-red-team/wiki.

[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] Gurock. Testrail. https://testrail.com/. Accessed: 2024-11-03.

[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] PyCQA. Pylint on pypi. https://pypi.org/project/pylint/. Accessed: 2024-11-03.

[9] 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

Minimalist Design

Please rate each statement on a scale of 1 to 5, where 1 = Strongly Disagree and 5 = Strongly Agree.

1. The tool's interface feels uncluttered, showing only essential elements.
2. I am able to focus on refactoring tasks without unnecessary distractions (rate 1-5).

Professional & Authoritative Appearance

Please rate each statement on a scale of 1 to 5, where 1 = Strongly Disagree and 5 = Strongly Agree.

1. The tool has a professional appearance that instills confidence in its functionality.
2. I would feel comfortable recommending this tool to other professionals based on its visual design.

Calm and Focused Atmosphere

Please rate each statement on a scale of 1 to 5, where 1 = Strongly Disagree and 5 = Strongly Agree.

1. The design of the tool creates a calm environment that supports my concentration on refactoring tasks.
2. The colours and layout used in the tool help reduce distractions and maintain my focus.

Modern and Visually Appealing Design

Please rate each statement on a scale of 1 to 5, where 1 = Strongly Disagree and 5 = Strongly Agree.

1. The tool's design is modern and aligns well with contemporary software development tools.
2. The interface design is aesthetically pleasing and enjoyable to use.

Intuitive UI and Ease of Use

Please rate each statement on a scale of 1 to 5, where 1 = Strongly Disagree and 5 = Strongly Agree.

1. The tool's interface is intuitive and easy to navigate.
2. I can quickly find key features and settings within the tool.

Clear and Concise Prompts

Please rate each statement on a scale of 1 to 5, where 1 = Strongly Disagree and 5 = Strongly Agree.

1. The prompts and instructions in the tool are clear and easy to understand.
2. The tool's prompts guide me effectively through processes.

Context-Sensitive Help

Please rate each statement on a scale of 1 to 5, where 1 = Strongly Disagree and 5 = Strongly Agree.

1. Help resources are easy to access and relevant to my current actions.
2. The tool provides useful help based on the task I am performing.

Clear and Constructive Error Messaging

Please rate each statement on a scale of 1 to 5, where 1 = Strongly Disagree and 5 = Strongly Agree.

1. The error messages in the tool are helpful and clearly explain the issue.
2. The tool provides constructive guidance on resolving errors I encounter.

Cultural sensitivity

1. What is your ethnicity or cultural background? (This question is optional and helps us understand how the tool is perceived across different cultures.)
    - African or African diaspora (e.g., African American, Afro-Caribbean)
    - East Asian (e.g., Chinese, Japanese, Korean)
    - South Asian (e.g., Indian, Pakistani, Bangladeshi)
    - Southeast Asian (e.g., Filipino, Vietnamese, Thai)
    - Middle Eastern or North African (MENA)
    - Hispanic or Latino/a
    - Indigenous or Native (e.g., Native American, First Nations, Aboriginal)

    - Pacific Islander
    - European or White/Caucasian
    - Mixed or Multi-ethnic
    - Prefer not to answer
    - Other (please specify): [Open text field]

2. Did you encounter any language, imagery, or content that felt insensitive?
    - No
    - Yes (Elaborate)

Overall Feedback

1. Are there any design improvements you would suggest? (optional)
2. What do you like most about the tool's design? (optional)

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.