System Verification and Validation Plan for Software Engineering

Team 4, EcoOptimizers

Nivetha Kuruparan
Sevhena Walker
Tanveer Brar
Mya Hussain
Ayushi Amin

April 5, 2025

Revision History

Date	Name	Notes
November 4th, 2024	All	Created initial revision of VnV Plan
January 3rd, 2025	Sevhena Walker	Modified template for static tests, clarified test-SRT-3
March 10th, 2025	Nivetha Kuruparan, Sevhena Walker	Revised Functional and Non-Functional Requirements
April 1st, 2025	Sevhena Walker	Modified Implementation Verification plan: refined unit testing tools.
April 1st, 2025	Sevhena Walker	Updated Automated testing and Verification Tools to include plugin tools.
April 1st, 2025	Sevhena Walker	Updated 4.2 and 4.5.2 to reflect the new tests.
April 1st, 2025	Sevhena Walker	Fixed some links. Fixed table formatting and some grammar.
April 3rd, 2025	Nivetha Kuruparan	Major Revisions for General Information Section
April 3rd, 2025	Nivetha Kuruparan	Major Revisions for Plan Section
April 3rd, 2025	Nivetha Kuruparan	Major Revisions for Test Functional Requirements Section
April 3rd, 2025	Nivetha Kuruparan	Major Revisions for Test Non-Functional Requirements Section
April 3rd, 2025	Nivetha Kuruparan	Heavily Revised VSCode Plugin Unit Tests
April 3rd, 2025	Nivetha Kuruparan	Fixed Links

Contents

List of Tables iv

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

2 Plan 3
- 2.1 Verification and Validation Team . . . 3
- 2.2 SRS Verification Plan . . . 3
  - 2.2.1 Requirements Validation . . . 3
  - 2.2.2 Stakeholder Review . . . 4
  - 2.2.3 User Acceptance Testing . . . 4
  - 2.2.4 Continuous Verification . . . 4
- 2.3 Design Verification Plan . . . 5
- 2.4 Verification and Validation Plan Verification Plan . . . 6
- 2.5 Implementation Verification Plan . . . 7
- 2.6 Automated Testing and Verification Tools . . . 8
- 2.7 Software Validation Plan . . . 9

3 System Tests 10
- 3.1 Tests for Functional Requirements . . . 10
  - 3.1.1 Code Input Acceptance Tests . . . 10
  - 3.1.2 Code Smell Detection Tests and Refactoring Suggestion (RS) Tests . . . 11
  - 3.1.3 Tests for Reporting Functionality . . . 12
  - 3.1.4 Visual Studio Code Interactions . . . 13
  - 3.1.5 Documentation Availability Tests . . . 13
- 3.2 Tests for Nonfunctional Requirements . . . 14
  - 3.2.1 Look and Feel . . . 14
  - 3.2.2 Usability & Humanity . . . 15
  - 3.2.3 Performance . . . 17
  - 3.2.4 Operational & Environmental . . . 18
  - 3.2.5 Maintenance and Support . . . 19
  - 3.2.6 Security . . . 20
  - 3.2.7 Cultural . . . 21
  - 3.2.8 Compliance . . . 21
- 3.3 Traceability Between Test Cases and Requirements . . . 23

4 Unit Test Description 25
- 4.1 Detection Module . . . 25
  - 4.1.1 Analyzer Controller . . . 25
  - 4.1.2 String Concatenation in a Loop . . . 26

4.1.3 Long Element Chain . . . 27
4.1.4 Repeated Calls . . . 28
4.1.5 Long Message Chain . . . 29
4.1.6 Long Lambda Element . . . 31
4.2 CodeCarbon Measurement Module . . . 32
4.3 Refactoring Module . . . 33
4.3.1 Refactorer Controller . . . 33
4.3.2 String Concatenation in a Loop . . . 34
4.3.3 Long Element Chain . . . 35
4.3.4 Member Ignoring Method . . . 36
4.3.5 Use a Generator . . . 37
4.3.6 Cache Repeated Calls . . . 38
4.3.7 Long Parameter List . . . 39
4.3.8 Long Message Chain . . . 40
4.3.9 Long Lambda Element . . . 41
4.4 VsCode Plugin . . . 42
4.4.1 Configure Workspace Command . . . 42
4.4.2 Reset Configuration Command . . . 43
4.4.3 File and Folder Smell Detection Commands . . . 44
4.4.4 Export Metrics Command . . . 45
4.4.5 Filter Smell Command Registration . . . 46
4.4.6 Refactor Workflow Commands . . . 47
4.4.7 Wipe Workspace Cache Command . . . 48
4.4.8 Workspace Modified Listener . . . 49
4.4.9 Backend Service Communication . . . 50
4.4.10 File Highlighter . . . 51
4.4.11 Hover Manager . . . 51
4.4.12 Line Selection Manager . . . 52
4.4.13 Cache Initialization . . . 53
4.4.14 Smells Data Management . . . 53
4.4.15 Tracked Diff Editors . . . 54
4.4.16 Refactor Action Buttons . . . 55

References **55**

Appendices **57**

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

B Reflection **58**

List of Tables

1 SRS Verification Checklist 5
2 Design Verification Checklist 6
3 V&V Plan Verification Checklist 7
4 Implementation Verification Checklist 8
5 Functional Requirements and Corresponding Test Sections 23
6 Look & Feel Tests and Corresponding Requirements 23
7 Usability & Humanity Tests and Corresponding Requirements 23
8 Performance Tests and Corresponding Requirements 23
9 Operational & Environmental Tests and Corresponding Requirements 24
10 Maintenance & Support Tests and Corresponding Requirements 24
11 Security Tests and Corresponding Requirements 24
12 Compliance Tests and Corresponding Requirements 24

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 Visual Studio Code (VS Code) extension designed to help developers identify and refactor energy-inefficient code in Python. It uses a Python package as its backend to detect code smells and estimate the carbon emissions that can be saved by applying the suggested refactorings. Users are given the option to accept or reject these refactorings directly within the editor. EcoOptimizer aims to promote energy-aware software development by making it easier to write efficient code without altering the original functionality.

1.2 Objectives

The primary objective of this project is to build confidence in the correctness and energy efficiency of the EcoOptimizer system, ensuring it reliably identifies and refactors energy-inefficient Python code without altering its original behaviour. Usability is a major focus—particularly within the VS Code extension—as the tool is designed to suggest improvements and keep developers informed while they write code. A core feature of EcoOptimizer is giving users the autonomy to accept or reject refactorings, empowering them to make informed decisions about the changes being applied to their code. These qualities—correctness, energy efficiency, user control, and usability—are essential to the project's success, as they shape the overall user experience, practical adoption, and sustainable benefits of the tool.

Certain objectives have been intentionally excluded due to resource constraints. For instance, we will not independently verify third-party libraries or dependencies, and will assume that they are adequately tested and maintained by their original developers.

1.3 Challenge Level and Extras

Our project, set at a general challenge level, includes two additional focuses: the creation of a user manual and the completion of a usability report. The user manual provides clear and accessible instructions for developers, covering installation, setup, and usage of both the

VS Code extension and the underlying Python package. It is designed to help users quickly integrate the tool into their development workflow. The usability report summarizes findings from a structured usability testing session, offering valuable insights into how effectively the tool meets user needs. These findings are used to refine the user interface and improve the overall user experience, ensuring the tool is both intuitive and practical for real-world use.

1.4 Relevant Documentation

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

Software Requirements Specification (SRS) [6]: 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) [4]: 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) [5]: 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 EcoOptimizer 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

The verification of the Software Requirements Specification will be conducted through a structured four-phase approach encompassing requirements validation, stakeholder review, user acceptance testing, and continuous verification.

2.2.1 Requirements Validation

The validation process will begin with comprehensive test coverage of all functional and non-functional requirements. A suite of test cases will be developed, employing automated testing frameworks such as PyTest [2] for functional requirements while reserving manual testing for edge cases and complex scenarios. To ensure complete traceability, we will maintain a matrix that explicitly links each requirement to its corresponding implementation, test cases, and test results.

2.2.2 Stakeholder Review

A formal review session will be conducted with our project supervisor, Dr. Istvan David, following a structured agenda. Prior to the meeting, we will prepare and distribute a package containing: (1) a summary of key requirements and design decisions, (2) visual aids including requirement diagrams and tables, and (3) specific questions regarding high-risk areas. During the session, we will conduct a detailed walkthrough of critical system components, particularly focusing on the energy measurement module and refactoring logic.

2.2.3 User Acceptance Testing

The verification process will include rigorous user acceptance testing involving 5–10 representative developers from our target user base. These tests will validate both the usability requirements (ensuring tasks can be completed within the **MAX_TASK_CLICKS** threshold of 4) and the effectiveness of energy metrics presentation (as specified in **FR6**). Feedback from these sessions will be systematically analyzed to identify any discrepancies between the SRS and actual user expectations.

2.2.4 Continuous Verification

To maintain alignment between the SRS and evolving system implementation, we will conduct biweekly review sessions. These sessions will serve to:

- Assess the impact of any requirement changes
- Update documentation to reflect system modifications
- Verify that all changes maintain consistency with the original specifications

Table 1: SRS Verification Checklist

Phase	**Verification Activities**
Requirements Validation	☐ Develop comprehensive test suite ☐ Conduct automated and manual testing ☐ Maintain traceability matrix
Stakeholder Review	☐ Prepare review materials ☐ Conduct formal walkthrough ☐ Address high-risk concerns
User Acceptance Testing	☐ Recruit representative users ☐ Validate usability metrics ☐ Analyze feedback
Continuous Verification	☐ Conduct biweekly reviews ☐ Maintain change documentation

2.3 Design Verification Plan

The design verification will focus exclusively on validating the system architecture and implementation specifications through structured peer and supervisor reviews. Classmates will conduct checklist-driven evaluations of the design document, paying particular attention to the refactoring options and the VS Code extension's interface design. Feedback will be consolidated in GitHub Issues and addressed in design refinement meetings.

A formal review with Dr. Istvan David will verify three critical aspects: (1) the energy measurement module's integration, (2) compliance with VS Code extension best practices, and (3) the fault-tolerance of the refactoring pipeline. We will prepare UML sequence diagrams and component specifications to facilitate this technical discussion, documenting all action items in our project board with clear resolution deadlines.

Table 2: Design Verification Checklist

Focus Area	**Verification Tasks**
Core Architecture	☐ Refactoring engine modularity confirmed
IDE Integration	☐ VS Code API usage reviewed ☐ UI/UX patterns verified
Data Integrity	☐ Energy measurement accuracy checked ☐ Code transformation safety ensured

2.4 Verification and Validation Plan Verification Plan

The Verification and Validation Plan for EcoOptimizer will be verified through peer reviews and targeted testing strategies. Team members and classmates will evaluate the plan's completeness using a structured checklist, focusing specifically on coverage of Python refactoring scenarios and energy efficiency measurements. Feedback will be tracked through GitHub Issues and incorporated in weekly team meetings.

For test effectiveness validation, we will employ mutation testing by introducing faults in sample code containing energy-inefficient patterns. This will quantitatively verify our test cases' ability to detect anomalies. The verification process will measure three key metrics: requirement coverage percentage, test case effectiveness, and issue resolution rate.

Table 3: V&V Plan Verification Checklist

Criteria	**Verification Tasks**
Coverage	☐ All refactoring cases included ☐ Energy metrics validation specified
Methodology	☐ Appropriate test levels defined ☐ Fault detection strategy in place
Process	☐ Feedback mechanism established ☐ Tracking system implemented

An iterative refinement process will be implemented, where verification findings are documented as GitHub issues and addressed in sprint reviews. This ensures the V&V plan remains aligned with both the technical requirements of Python code optimization and the project's timeline constraints. Progress will be measured against predefined success metrics, including test coverage percentages and mutation detection rates.

2.5 Implementation Verification Plan

The implementation verification will ensure EcoOptimizer's codebase strictly adheres to the SRS specifications through rigorous testing protocols. Unit testing will validate core functionality using Pytest [2] for Python components (refactoring engine, energy measurement) and Jest [7] for the VS Code extension (UI integration). Test cases will specifically target energy-efficient transformations identified in **FR2-FR4**.

Static analysis will enforce code quality using Python linters to verify compliance with PEP 8 [10] standards (per **CR-SCR1**) and detect security vulnerabilities. Weekly peer code reviews will examine implementation quality, focusing on:

- Algorithm efficiency for energy optimization
- Correct handling of Python version-specific features **(MS-AD3**)
- VS Code extension usability (**UHR-EOU1**)

Performance testing will validate:

- Refactoring speed against **PR-SL1** thresholds
- Energy measurement accuracy (**FR6**)
- Large codebase handling (**PR-CR1**)

Table 4: Implementation Verification Checklist

Category	Verification Tasks
Functionality	☐ Unit tests for all refactoring methods ☐ Energy measurement validation
Quality	☐ Static analysis completed ☐ Code reviews conducted
Performance	☐ Processing time benchmarks ☐ Large codebase tests

2.6 Automated Testing and Verification Tools

Unit Testing Framework: The project uses standard testing tools for each part of the system: `Pytest` [2] for the Python backend and `Jest` [7] for the TypeScript frontend. These tools were chosen because they are widely used in their respective ecosystems, well-documented, and compatible with the project's CI/CD pipeline. Together they provide test coverage for both backend and frontend components.

Code Coverage Tools and Plan for Summary: The codebase will be analyzed to determine the percentage of code executed during tests using language-specific tools:

- **Python:** `pytest-cov` [3] provides granular-level coverage including branch, line, and path analysis. Its seamless integration with `Pytest` [2] ensures coverage metrics are generated during normal test execution.
- **TypeScript:** `Jest`'s [7] built-in coverage functionality will track statement, branch, and function coverage for the VS Code extension, with configuration matching the Python tool's output format.

Initially, the aim is to achieve 40% coverage across both codebases, gradually incrementing the level over time. Weekly reports generated from both tools will be combined to track coverage trends, identify testing gaps, and set improvement goals as the project evolves. The unified coverage data will ensure consistent quality standards are maintained throughout the full stack.

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

`Prettier` [9] for the TypeScript extension.

Testing Strategy for the VSCode Extension: The TypeScript extension will be tested using `Jest` [7]. Automated tests will verify interactions between the extension and the editor, reducing regressions during development.

2.7 Software Validation Plan

EcoOptimizer will be validated through:

- **Functional Testing**: Evaluation against open-source Python projects to verify:
  - Energy efficiency improvements (**FR3**, **FR6**)
  - Refactoring accuracy (**FR4**)
- **Usability Testing**: Sessions with Dr. David and Python developers assessing:
  - VS Code extension workflow (**UHR-EOU1-2**)
  - Developer experience metrics
- **Formal Review**: Rev 0 demo with Dr. David to validate:
  - SRS requirement implementation
  - System behavior against specifications

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 [6]. 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 (**FR1**).

test-FR-IA-1 Valid Python File Acceptance

Control: Manual
Initial State: Tool is idle within the VS Code workspace.
Input: A valid Python file (filename.py) containing syntactically correct code.
Output: The system accepts the file without errors and displays any detected code smells if present.

Test Case Derivation: Confirming that the system correctly processes valid Python files in the supported environment, as specified in **FR1**.

How test will be performed: Open a syntactically valid '.py' file in VS Code with the extension enabled. Verify that the tool processes the file and optionally displays code smells if any are present.

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

Control: Manual
Initial State: Tool is idle within the VS Code workspace.
Input: A '.py' file (badSyntax.py) containing deliberate syntax errors that prevent parsing.
Output: The system detects the issue, halts further analysis, and displays an appropriate error message within the editor.

Test Case Derivation: Ensures graceful handling of invalid Python syntax and appropriate user feedback, satisfying **FR1**.

How test will be performed: Load a Python file with syntax errors in VS Code, then observe whether the extension flags the syntax issue and stops further processing.

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

Control: Manual
Initial State: Tool is idle within the VS Code workspace.
Input: A non-Python file (e.g., ‘notes.txt‘ or ‘script.js‘).
Output: The system ignores the file or displays a message indicating that the file type is unsupported.

Test Case Derivation: Validates that the system filters non-Python files, consistent with the requirement that it must exclusively process ‘.py‘ files (**FR1**).

How test will be performed: Attempt to open a non-Python file in VS Code and check that the extension does not attempt analysis or refactoring and provides clear messaging if applicable.

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

---

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

test-FR-IA-1 Successful Refactoring Execution

Control: Automated
Initial State: Tool is idle in the VS Code environment.
Input: A valid Python file with a detectable code smell.
Output: The system applies the appropriate refactoring and updates the code view.

Test Case Derivation: Ensures the tool correctly identifies a smell (e.g., LEC001), chooses an applicable refactoring, and applies it successfully, per **FR2** and **FR3**.

How test will be performed: Provide a valid Python file containing a known smell, trigger refactoring via the VS Code interface, and confirm the output includes refactored code as expected.

test-FR-IA-2 No Available Refactorer Handling

Control: Automated
Initial State: Tool is idle.
Input: A valid Python file containing a code smell that does not yet have a supported refactorer.
Output: The system does not apply changes and logs or displays an informative message.

Test Case Derivation: Verifies that unsupported code smells are gracefully handled without errors, per **FR2**.

How test will be performed: Provide a valid Python file with an unsupported smell and observe that the system notifies the user without attempting modification.

test-FR-IA-3 Multiple Refactoring Calls on Same File

Control: Automated
Initial State: Tool is idle.
Input: A valid Python file with a detectable code smell, refactored more than once.
Output: The tool processes the file repeatedly and applies changes incrementally.

Test Case Derivation: Confirms the system can handle repeated invocations and re-apply applicable refactorings, per **FR3**.

How test will be performed: Refactor a file containing a supported smell multiple times and verify that each run performs valid operations and results in updated outputs.

test-FR-IA-4 Handling Empty Modified Files List

Control: Automated
Initial State: Tool is idle.
Input: A valid Python file where the code smell is detected, but the refactorer makes no modifications.
Output: The system does not generate output files and notifies the user appropriately.

Test Case Derivation: Confirms the tool handles no-op refactorers correctly, per **FR4**.

How test will be performed: Supply a file where the refactorer returns an unchanged version of the code and verify that no new files are created and that appropriate feedback is displayed or logged.

3.1.3 Tests for Reporting Functionality

The reporting functionality of the tool is crucial for providing users with meaningful insights into the energy impact of refactorings and the smells being addressed. This section outlines tests that ensure the energy metrics and refactoring summaries are accurately presented, as required by **FR6** and **FR15**.

test-FR-RP-1 Energy Consumption Metrics Displayed Post-Refactoring

Control: Manual
Initial State: The tool has measured energy usage before and after refactoring.
Input: Energy data collected for the original and refactored code.
Output: A clear comparison of energy consumption is displayed in the UI.

Test Case Derivation: Verifies that energy metrics are properly calculated and presented to users, as per **FR6**.

How test will be performed: Refactor a file and review the visual or textual display of energy usage before and after, ensuring the values match backend logs.

test-FR-RP-2 Detected Code Smells and Refactorings Reflected in UI

Control: Manual
Initial State: The tool has completed code analysis and refactoring.
Input: Output of the detection and refactoring modules.
Output: The user interface displays the detected code smells and associated refactorings clearly.

Test Case Derivation: Ensures transparency of changes and supports informed decision-making by the user, in line with **FR15**.

How test will be performed: Open a code file with detectable smells, trigger a refactor, and inspect the view displaying the summary of changes and available actions.

3.1.4 Visual Studio Code Interactions

This section corresponds to features related to the user's interaction with the Visual Studio Code extension interface, including previewing and toggling smells, customizing the UI, and reviewing code comparisons. These tests verify that the extension enables users to interact with refactorings in an intuitive and informative manner, as outlined in **FR8**, **FR9**, **FR10**, **FR11**, **FR12**, **FR13**, **FR14**, **FR15**, **FR16**, and **FR17**.

These features are primarily tested through automated unit tests integrated in the extension codebase. For implementation and test details, please refer to the unit testing suite.

3.1.5 Documentation Availability Tests

The following test is designed to ensure the availability of documentation as per **FR 7** and **FR 5**.

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 (**FR 5**), and troubleshooting.

Test Case Derivation: Validates that the documentation meets user needs (**FR 7**).

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

3.2 Tests for Nonfunctional Requirements

The section will cover system tests for the non-functional requirements (NFR) listed in the SRS document[6]. 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.

For non-functional requirements that are not linked to a test in the below sections, has either been covered in the functional requirements test plan or unit tests plan. Please see traceability matrix for more details.

3.2.1 Look and Feel

The following subsection tests cover all Look and Feel non-functional requirements listed in the SRS [6]. They aim to validate that the system provides a modern, visually appealing, and supportive developer experience. These tests ensure that the tool facilitates refactoring decisions through clear interfaces and satisfies design expectations based on user perception.

test-LF-1 Side-by-side Code Comparison in IDE Plugin

Type: Non-Functional, Manual, Dynamic
Covers: LFR-AP1
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 file and apply a refactoring. They will verify that the original and refactored code are shown side by side with functional accept/reject buttons. This confirms users can make informed refactoring decisions in a visually supportive layout.

test-LF-2 Design Acceptance Survey

Type: Non-Functional, Manual, Dynamic
Covers: LFR-ST1, *implicitly covers* LFR-AP2
Initial State: IDE plugin open
Input/Condition: Developer interacts with the plugin
Output/Result: A survey response capturing the user's perception of the design

How test will be performed: After using the plugin, test participants will complete the design satisfaction survey described in A.2. The results will be reviewed to assess whether the plugin meets the project's aesthetic and usability expectations.

Note: LFR-AP2 is not tested directly, as it describes a design philosophy rather than a testable behavior. User perception of visual simplicity and minimalism is instead captured through feedback in the usability survey.

3.2.2 Usability & Humanity

The following tests cover all Usability & Humanity requirements listed in the SRS [6]. These tests aim to validate that the system is accessible, user-centred, intuitive, and easy to navigate. Data is collected via user surveys or static analysis, and evaluated against thresholds (e.g., 80–90% agreement). Where applicable, tests are traceable to corresponding SRS requirements.

test-UH-1 Customizable Settings for Refactoring Preferences

Type: Manual, Dynamic
Covers: UHR-PSI 1 & **UHR-PSI 2**
Initial State: Plugin open with settings panel accessible
Input/Condition: User customizes enabled smells and highlight colors
Output/Result: Preferences persist and affect plugin behavior

How test will be performed: Tester toggles smell types and changes highlight colours. They reload the plugin and verify that settings are retained and correctly reflected in UI and detection.

test-UH-2 High-Contrast Theme Accessibility Check

Type: Static Analysis
Covers: UHR-ACS 1
Initial State: Contrast-based theme styles configured
Input/Condition: Contrast analyzer is run on UI color tokens
Output/Result: All items meet 4.5:1 or 3:1 WCAG thresholds

How test will be performed: Use automated tools (e.g., WebAIM) to verify foreground/background ratios for code highlights, sidebars, and messages.

test-UH-3 Intuitive User Interface for Core Functionality

Type: User Testing, Survey-Based
Covers: UHR-EOU 1
Initial State: Plugin open with test tasks prepared
Input/Condition: Users complete core tasks (detect, refactor, configure)
Output/Result: 90% of users complete each task in ≤ 3 clicks and rate the interaction as intuitive

How test will be performed: Clicks per task are recorded. After each task, users answer the question "Was this process intuitive?" on a 5-point Likert scale. The question is listed in Appendix A.2.
Quantifiable Metric: At least 85% of responses must score 4 or 5 on the intuitiveness scale.

Use of Results: Responses scoring below threshold will trigger UX redesign of the corresponding interaction. Open feedback (if given) will be coded thematically to identify patterns in confusion or friction points.

test-UH-4 Clear and Concise User Prompts

Type: Survey-Based
Covers: UHR-EOU 2
Initial State: User encounters plugin prompts (e.g., file missing, confirm refactor)
Input/Condition: Users follow prompts and evaluate clarity
Output/Result: 90% of users agree prompts are helpful and unambiguous

How test will be performed: After interacting with all major system prompts, users complete a survey (Appendix A.2) where they rate the clarity of each message on a 5-point scale and provide optional comments.
Quantifiable Metric: 90% of users must rate each prompt ≥ 4 for clarity and helpfulness.
Use of Results: Any prompt scoring below target will be reviewed and rewritten. Free-text feedback will be grouped by theme to identify language, formatting, or placement issues.

test-UH-5 Context-Sensitive Help Based on User Actions

Type: Manual
Covers: UHR-LRN 1
Initial State: Help system available via command/hover
Input/Condition: User performs smell-related actions, requests help
Output/Result: Help shown is contextually relevant, appears within <= 3 clicks

How test will be performed: Tester performs tasks like "Apply refactor" or "Toggle smell" and verifies help popups match the current feature.

test-UH-6 Clear and Constructive Error Messaging

Type: Survey-Based
Covers: UHR-UPL 1
Initial State: Plugin triggers common errors (e.g., no workspace configured, backend offline)
Input/Condition: User encounters error messages during usage and evaluates tone and clarity
Output/Result: 80% of users agree the message is polite, understandable, and helpful

How test will be performed: After encountering various plugin error scenarios, users complete a survey located in Appendix A.2. They rate each error message on tone, clarity, and helpfulness using a 5-point Likert scale, and may provide free-text feedback.
Quantifiable Metric: Each error message must receive an average rating of ≥ 4 in tone, clarity, and helpfulness from at least 80% of participants.
Use of Results: Messages scoring below threshold will be flagged for revision. Qualitative responses will be categorized to identify recurring issues such as overly technical language, lack of actionable steps, or negative tone.

3.2.3 Performance

The following subsection tests cover the Performance requirements listed in the SRS [6]. The goal is to validate that the tool can process Python files of varying sizes within acceptable performance thresholds. These tests confirm responsiveness under real-world usage, and guide profiling and optimization work for scalability.

test-PF-1 Performance and Capacity Validation for Analysis and Refactoring

Type: Non-Functional, Automated, Dynamic
Covers: PR-SL 1, PR-SL 2, PR-CR 1
Initial State: IDE open with multiple Python files of varying sizes prepared (small: 250 LOC, medium: 1000 LOC, large: 3000 LOC)
Input/Condition: Initiate detection and refactoring for each file sequentially
Output/Result: Tool completes within:

- 20 seconds for small files ($\leq$ 250 lines)
- 50 seconds for medium files ($\leq$ 1000 lines)
- 2 minutes for large files ($\leq$ 3000 lines)

How test will be performed: Detection and refactoring will be run on each file. Timings will be recorded from start to finish. If thresholds are exceeded, logs and profiling output will be used to identify and prioritize optimization targets.

Untested Requirements and Justification:

- **PR-SCR 1 (No runtime errors in refactored code)**: Verified by functional unit tests and integration tests that execute refactored code and ensure correctness and compilability.
- **PR-PAR 1 (Smell detection accuracy)**: Already covered in functional tests, which compare the detected smells against a ground truth dataset and calculate precision/recall.
- **PR-PAR 2 (Output validity)**: Confirmed by functional tests that ensure the generated refactored code is syntactically valid and matches Python standards.
- **PR-RFT 1 (Robustness to invalid input)**: Addressed through functional tests which simulate corrupt files or invalid syntax and assert the system recovers gracefully.
- **PR-SER 1 (Extensibility)**: Verified through manual inspection and code review of plugin architecture and smell registration pipeline; extensibility is not runtime-measurable and therefore not performance tested.

- **PR-LR 1 (Longevity)**: Also verified through code quality reviews. Maintainability is supported by documentation and modularity but not measurable via direct tests in this release cycle.

3.2.4 Operational & Environmental

The following subsection tests cover all Operational and Environmental requirements listed in the SRS [6]. This includes confirming system compatibility, installation capability, and basic usability across operational contexts. Physical environment requirements are not tested, as explained below.

test-OPE-1 VS Code Compatibility for Refactoring Library Extension

Type: Non-Functional, Manual, Dynamic
Covers: OER-IAS 1
Initial State: VS Code IDE open and library not installed
Input/Condition: User installs and opens the refactoring library extension in VS Code
Output/Result: The extension installs successfully and runs inside VS Code

How test will be performed: The tester will search for the extension on the VS Code marketplace, install it, and verify that it appears in the IDE and can execute basic functionality such as detecting or refactoring a code file.

test-OPE-2 Import and Export Capabilities for Codebases and Metrics

Type: Non-Functional, Manual, Dynamic
Covers: OER-IAS 2
Initial State: IDE plugin open with option to import/export
Input/Condition: User imports a sample project and exports results
Output/Result: Plugin correctly imports project and generates JSON/XML reports for refactored code and metrics

How test will be performed: Tester loads a sample codebase, initiates a refactor, and uses the export function. Output files are verified for valid structure and meaningful content.

test-OPE-3 PIP Package Installation Availability

Type: Non-Functional, Manual, Dynamic
Covers: OER-PR 1
Initial State: Fresh Python environment without the package
Input/Condition: User runs `pip install ecooptimizer`
Output/Result: The package installs successfully without error

How test will be performed: Tester uses a clean Python virtual environment to install the library. A short script using a sample function will verify it works as expected after installation.

Unnecessary Tests and Justifications

- **OER-EP 1 & OER-EP 2 (Temperature & Power Requirements)**:
  These describe expected hardware operating conditions. If the computer cannot operate due to temperature or power issues, the software cannot run. These are not properties of the software and are thus *not tested* in this V&V plan.
- **OER-RL 1 (All core functionality implemented and tested)**:
  This requirement is satisfied by the completion of the full functional and non-functional test suite. No specific test case is needed beyond traceability to those tests.
- **OER-RL 2 (Release by March 17, 2025)**:
  This is a delivery milestone tracked through project management, not through testing. Therefore, it is *not verified dynamically* through test cases.

3.2.5 Maintenance and Support

The following subsection tests cover the most critical Maintenance and Support requirements listed in the SRS [6]. These tests emphasize extensibility, maintainability, and recovery from faulty updates. Some lower-priority requirements are acknowledged but excluded from testing due to scope or redundancy with existing CI/CD practices.

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

Covers: MS-MNT 1
Type: Code Walkthrough and Manual Dynamic
Initial State: Developer environment set up with project codebase
Input/Condition: Developer adds a sample code smell and refactoring method
Output/Result: The new smell/refactoring integrates with minimal disruption

How test will be performed: Developers will follow documentation to add a new smell and refactoring function. They will verify modularity and confirm that existing functionality is unaffected. Success is defined by clean integration and interface visibility without needing major code changes.

test-MS-2 **Maintainable and Adaptable Codebase**

Covers: MS-MNT 2
Type: Static Analysis and Documentation Review
Initial State: Final implementation and documentation available
Input/Condition: Reviewers evaluate code organization and documentation
Output/Result: Codebase is modular, readable, and sufficiently documented

How test will be performed: Reviewers examine file structure, naming conventions, and presence of comments. They will evaluate whether a new developer could easily navigate and modify the code. Documentation completeness will be scored on a checklist.

test-MS-3 Rollback Support for Faulty Updates

Covers: MS-MNT 3
Type: Manual Dynamic
Initial State: Installed version of the library in use
Input/Condition: Faulty update is simulated; rollback is triggered
Output/Result: System returns to previous stable state successfully

How test will be performed: A new version with an intentional fault is deployed. The user will invoke rollback steps (e.g., Git revert or VS Code extension downgrade). The tool should resume normal operation with no feature regressions.

Unnecessary Tests and Justifications

- **MS-MNT 4 (Automated Testing for Refactorings):** Verified through existing unit tests and CI pipeline. Manual test duplication is unnecessary.
- **MS-MNT 5 (Library Compatibility with Dependencies):** Assumed validated during integration testing and package dependency resolution via PIP.

3.2.6 Security

The following subsection tests cover the primary Security requirement listed in the SRS [6]. These tests ensure that refactoring operations are traceable, logged securely, and protected from unauthorized tampering. Due to the tool's local-only design, network-level security and external access controls are out of scope.

test-SRT-1 Audit Logs for Refactoring Processes

Covers: SRT-AUD 1
Type: Static Analysis and Code Review
Initial State: Fully implemented refactoring pipeline with logging enabled
Input/Condition: Refactoring actions performed on one or more files
Output/Result: Secure and complete logs are generated for each action

How test will be performed: The development team will review logging logic in the code to ensure it covers all major events: pattern detection, energy analysis, and refactor generation. Each entry must include a timestamp, file reference, and action type. Reviewers will confirm that logs are immutable and not user-editable, using tools like append-only file systems or cryptographic checksums if applicable.

3.2.7 Cultural

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

3.2.8 Compliance

The following subsection tests cover all Compliance requirements listed in the SRS [6]. These tests ensure that the system does not collect personal user data and that the codebase adheres to established Python development standards.

test-CPL-1 User Privacy and Local Execution Compliance

Covers: CR-LR 1
Type: Static Review and Code Audit
Initial State: Tool installed and ready for inspection
Input/Condition: Reviewer inspects runtime behavior and data access patterns
Output/Result: No personal or user-specific data is collected or transmitted

How test will be performed: Review the codebase to confirm that:

- The tool does not collect or store personal information.
- No external API requests are made that could leak user data.
- All operations occur locally, including refactoring, logging, and energy analysis.

The reviewer will also confirm the absence of telemetry or usage tracking modules.

Acceptance Criteria: The tool operates entirely locally, without collecting or transmitting any personal data. Privacy is preserved by design.

test-CPL-2 PEP 8 Standards Compliance

Covers: CR-SCR 1
Type: Static Analysis
Initial State: Codebase fully implemented
Input/Condition: Code is scanned using a PEP 8 linter (e.g., `flake8`, `pylint`)
Output/Result: All code conforms to Python PEP 8 coding standards

How test will be performed: Run a static code analysis tool across the full Python codebase. Evaluate:

- Adherence to line length, indentation, and naming conventions.

- Documentation comments and docstring usage.
- Overall maintainability and readability.

Any issues will be fixed and re-evaluated.

Acceptance Criteria: The code passes static analysis with no critical PEP 8 violations. Style warnings are minimized, and documentation is consistent.

3.3 Traceability Between Test Cases and Requirements

Table 5: Functional Requirements and Corresponding Test Sections

Test Section	**Functional Requirement(s)**
Code Input Acceptance Tests	FR1
Code Smell Detection and Refactoring Suggestion Tests	FR2, FR3, FR4
Tests for Reporting Functionality	FR6, FR15
Visual Studio Code Interactions	FR8, FR9, FR10, FR11, FR12, FR13, FR14, FR15, FR16, FR17
Documentation Availability Tests	FR7, FR5
Installation and Onboarding Tests	FR7

Table 6: Look & Feel Tests and Corresponding Requirements

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

Table 7: Usability & Humanity Tests and Corresponding Requirements

Test ID (test-)	**Non-Functional Requirement**
UH-1	UHR-PSI 1, UHR-PSI 2
UH-2	UHR-ACS 1
UH-3	UHR-EOU 1
UH-4	UHR-EOU 2
UH-5	UHR-LRN 1
UH-6	UHR-UPL 1

Table 8: Performance Tests and Corresponding Requirements

Test ID (test-)	**Non-Functional Requirement**
PF-1	PR-SL 1, PR-SL 2, PR-CR 1

Table 9: 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 10: 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 11: Security Tests and Corresponding Requirements

Test ID (test-)	Non-Functional Requirement
SRT-1	SR-IM 1

Table 12: Compliance Tests and Corresponding Requirements

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

4 Unit Test Description

4.1 Detection Module

4.1.1 Analyzer Controller

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

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

Target requirement(s): FR2, FR5 [6]

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

The test cases for this module can be found here.

4.1.2 String Concatenation in a Loop

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

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

Target requirement(s): FR2 [6]

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

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

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

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

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

The test cases for this module can be found here.

4.1.3 Long Element Chain

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

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

Target requirement(s): FR2 [6]

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

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

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

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

The test cases for this module can be found here.

4.1.4 Repeated Calls

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

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

Target requirement(s): FR2, FR3, PR-PAR2, PR-PAR3 [6]

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

The test cases for this module can be found here.

4.1.5 Long Message Chain

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

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

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

Target requirement(s): FR2 [6]

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

The test cases for this module can be found here

4.1.6 Long Lambda Element

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

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

Target requirement(s): FR2 [6]

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

- **Trivially short lambda function**
  - Verifies that degenerate cases, such as a lambda with no real body or trivial operations, are not mistakenly flagged

The test cases for this module can be found here

4.2 CodeCarbon Measurement Module

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

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

Target requirement(s): FR5, FR6,PR-RFT1, PR-SCR1 [6]

- **Successful Energy Measurement**
  - Verifies correct emissions tracking when CodeCarbon returns valid float values
  - Confirms proper subprocess execution and tracker API calls
  - Ensures emissions value is stored in the meter instance
- **Null Emissions Handling**
  - Validates system behavior when CodeCarbon returns None
  - Confirms meter stores None without errors
- **Unexpected Emissions Type Handling**
  - Tests proper logging and null conversion for non-float/non-None returns
  - Specifically verifies handling of string and NaN values
- **Subprocess Failure Resilience**
  - Confirms system logs errors but preserves emissions data when subprocess fails
  - Verifies tracker still stops properly after subprocess exceptions
- **CSV Data Extraction**
  - Validates correct parsing of multi-row emissions CSV files
  - Ensures last row is properly returned as current emissions

- **Missing Emissions File Handling**
  - Tests proper error logging when emissions file is missing
  - Verifies None is returned and stored in emissions_data

The test cases for this module can be found here.

4.3 Refactoring Module

4.3.1 Refactorer Controller

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

Target requirement(s): FR5, UHR-UPL1 [6]

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

The test cases for this module can be found here.

4.3.2 String Concatenation in a Loop

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

Target requirement(s): FR3, FR6 [6]

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

The test cases for this module can be found here.

4.3.3 Long Element Chain

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

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

Target requirement(s): FR3, FR6, PR-PAR3 [6]

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

The test cases for this module can be found here.

4.3.4 Member Ignoring Method

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

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

Target requirement(s): FR3, FR6 [6]

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

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

The test cases for this module can be found here.

4.3.5 Use a Generator

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

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

Target requirement(s): FR3, FR5, FR6, PR-PAR3 [6]

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

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

The test cases for this module can be found here.

4.3.6 Cache Repeated Calls

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

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

Target requirement(s): FR3, FR5, FR6, PR-PAR3, PR-PAR2 [6]

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

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

The test cases for this module can be found here.

4.3.7 Long Parameter List

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

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

Target requirement(s): FR3, FR6 [6]

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

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

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

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

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

The test cases for this module can be found here.

4.3.8 Long Message Chain

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

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

Target requirement(s): FR5, FR6, FR3 [6]

- **Basic method chain refactoring**
  - Ensures that a simple method chain is refactored correctly into separate intermediate variables.
- **F-string chain refactoring**
  - Verifies that method chains applied to f-strings are properly broken down while preserving correctness.
- **Modifications even if the chain is not long**
  - Ensures that method chains are refactored consistently, even if they do not exceed the length threshold.
- **Proper indentation preserved**
  - Confirms that the refactored code maintains the correct indentation when inside a block statement such as an `if` condition.
- **Method chain with arguments**
  - Tests that method chains containing arguments (e.g., `replace("H", "J")`) are correctly refactored.
- **Print statement preservation**
  - Ensures that method chains within a `print` statement are refactored without altering their functionality.
- **Nested method chains**
  - Verifies that nested method chains (e.g., method calls on method results) are properly refactored into intermediate variables.

The test cases for this module can be found here

4.3.9 Long Lambda Element

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

The goal of this test suite is to ensure long lambda expressions are refactored into named functions while maintaining code functionality, readability, and proper syntax. It verifies that simple single-line lambdas are converted correctly and that more complex cases, such as multi-line lambdas or those with multiple parameters, are handled appropriately. The tests ensure that refactoring preserves the original behavior of the code, even for lambdas used as keyword arguments or passed to functions like map or reduce. Additionally, the suite

confirms that no unnecessary changes, such as added print statements, are introduced during refactoring. By covering a wide range of cases, the suite ensures the refactoring process is both reliable and effective.

Target requirement(s): FR5, FR6, FR3 [6]

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

The test cases for this module can be found here

4.4 VsCode Plugin

4.4.1 Configure Workspace Command

Goal: The configure workspace command identifies valid workspace folders containing Python files and prompts the user to select one. Upon selection, the workspace is marked as configured and saved to persistent state for future operations.

The tests validate that valid workspace folders are detected, Python files are identified correctly, user selections are respected, and appropriate updates are made to both VS Code context and extension state.

Target requirement(s): FR1, FR2 [6]

- **Folder Scanning**
  - Detects top-level and nested directories containing `.py` files.
  - Correctly identifies directories with Python entry points (e.g., `main.py` or `__init__.py`).
- **Quick Pick Interaction**
  - Displays a list of valid Python folders.
  - Accepts user selection and confirms configuration.
- **Workspace State Update**
  - Stores selected folder path under the configured workspace key.
  - Sets VS Code context key `workspaceState.workspaceConfigured` to true.
- **Feedback to User**
  - Shows information message indicating selected folder.

The test cases for this module can be found here.

4.4.2 Reset Configuration Command

Goal: The reset configuration command prompts the user for confirmation and, if accepted, clears the stored workspace path and resets the internal plugin context to an unconfigured state.

The tests verify that the workspace state is properly reset only when the user confirms the action. The system must also update the relevant VS Code context key to reflect the unconfigured state.

Target requirement(s): FR3 [6]

- **User Confirmation**
  - Prompts user with a warning message before clearing configuration.
- **Workspace State Reset**
  - Clears the stored workspace path from extension state.
- **Context Reset**
  - Updates VS Code context key `workspaceState.workspaceConfigured` to false.

- **Cancel Handling**
  - Skips reset process if the user cancels the confirmation dialog.

The test cases for this module can be found here.

4.4.3 File and Folder Smell Detection Commands

Goal: The `detectSmellsFile` and `detectSmellsFolder` commands are responsible for initiating the detection of code smells in individual files and entire directories, respectively. These commands coordinate the interaction between the cache system, backend smell detection API, and the UI rendering logic within the SmellsViewProvider.

These unit tests ensure robust handling of various scenarios, including invalid input files, empty folders, disabled smell settings, cached data reuse, server downtime, backend errors, and recursive directory scans. They also validate that user-facing messages, caching, and smell highlighting are executed correctly based on system state.

Target requirement(s): FR2, FR3, FR15, OER-IAS2 [6]

- **Skipping Non-Python and Untitled Files**
  - The file detection command ignores non-Python files and unsaved/untitled documents.
- **Using Cached Smells**
  - Cached smells are reused when the file hash and settings match.
  - Cached smells are immediately rendered in the Smells View.
- **Fetching Smells from Backend**
  - When no cache is available, the backend is called to fetch smells.
  - Retrieved smells are stored in the cache and displayed in the UI.
- **Handling No Enabled Smells**
  - A warning message is shown when no smells are enabled in the configuration.
- **Handling No Smells Found**
  - The tool displays a message indicating that the file has no detectable smells.
  - The cache is updated to reflect the empty result.
- **Handling API and Server Errors**
  - Server-down conditions show appropriate warning messages.
  - API response failures result in error messages and a "failed" status in the UI.

  - Unexpected thrown exceptions are caught and logged to the output channel.

- **Folder Analysis and Recursion**
  - The folder command scans recursively to identify '.py' files and shows progress.
  - A warning is shown if no Python files are found in the directory.
  - The total number of analyzed files is reported to the user.

- **Handling Folder Access Errors**
  - Directory scan failures (e.g., permission errors) are caught and logged cleanly without crashing the extension.

The test cases for this module can be found here.

4.4.4 Export Metrics Command

Goal: The export metrics command saves workspace-level energy metrics to a local JSON file. This test suite ensures it handles missing data, invalid paths, file vs. directory distinction, and file system errors gracefully.

The tests validate correct behavior across different workspace path types, proper access to extension state, and informative error messaging for common failure scenarios.

Target requirement(s): FR6, FR15, OER-IAS3 [6]

- **No Metrics Data**
  - The command shows an information message if there is no metrics data available to export.

- **No Workspace Path Configured**
  - The command shows an error message if the workspace path is missing or unset.

- **Export to Workspace Directory**
  - If the workspace path is a directory, the metrics JSON is saved directly inside it.
  - The user is notified of the output file location.

- **Export to Parent of File Path**
  - If the workspace path is a file, the parent directory is used for export.

- **Invalid Path Type Handling**
  - The command shows an error message if the path type is unknown or unsupported.

- **Filesystem Access Errors**

- The command handles and reports errors when accessing the file system (e.g., stat failures).

- **File Write Failures**
  - If the write operation fails, the user is shown an error message indicating export failure.

The test cases for this module can be found here.

4.4.5 Filter Smell Command Registration

Goal: This command module registers all filter-related UI commands used to configure active smells and their parameters in the EcoOptimizer sidebar. The tests validate command registration, user interaction (e.g., input box), and the proper delegation to 'FilterViewProvider'.

The tests ensure that smell toggling, option editing, mass selection, and default resets all call the correct provider methods and handle edge cases such as missing inputs or invalid user values.

Target requirement(s): FR13, FR14, OER-IAS1 [6]

- **Command Registration**
  - Each command is correctly registered with VS Code using the expected command IDs.
  - All registered disposables are added to the extension context's subscriptions.
- **Toggling Individual Smells**
  - The command toggles a specific smell using `toggleSmell`.
- **Editing Smell Options – Valid Input**
  - The user is prompted for a new value via an input box.
  - The new numeric value is passed to `updateOption` and the filter view is refreshed.
- **Editing Smell Options – Invalid or Missing Input**
  - If the user enters a non-numeric value, the option is not updated.
  - If the smell or option key is missing, an error message is shown.
- **Select/Deselect All Smells**
  - The `selectAllFilterSmells` and `deselectAllFilterSmells` commands call `setAllSmellsEnabled(true/false)` respectively.
- **Reset to Filter Defaults**
  - The `setFilterDefaults` command calls `resetToDefaults` on the filter provider.

The test cases for this module can be found here.

4.4.6 Refactor Workflow Commands

Goal: The refactoring workflow includes the core commands responsible for initiating, applying, and discarding refactorings. These tests verify that backend communication, session setup, user feedback, and file handling work correctly across both individual and batch refactor scenarios.

This suite ensures refactor commands respect workspace configuration, gracefully handle backend issues, display energy savings, manage diff editors, and correctly update workspace state and metrics. Error handling for edge cases like file system failures and missing data is also validated.

Target requirement(s): FR3, FR4, FR15, OER-IAS4 [6]

- **Workspace and Server Preconditions**
  - Shows an error if the workspace is not configured.
  - Shows a warning if the backend server is unavailable.
- **Refactoring Execution – Single Smell**
  - Initiates refactoring for one smell using the backend.
  - Updates internal state and view providers.
  - Displays progress and success messages.
- **Refactoring Execution – All Smells of Same Type**
  - Calls bulk API endpoint for all smells of the given type.
  - Displays type-based success messages.
- **Refactoring Failure Handling**
  - Shows an error if the backend fails or throws.
  - Resets UI state and hides any open diff editors.
- **Session Initialization via `startRefactorSession`**
  - Opens a VS Code diff editor with original and refactored files.
  - Displays energy savings in an information message.
  - Handles missing energy savings values gracefully (e.g., N/A).
- **Accepting Refactorings**
  - Copies refactored files into the workspace.
  - Updates metrics data and clears smell cache for affected files.
  - Sets file status to "outdated" in the UI.

  - Handles missing session data or filesystem errors.

- **Rejecting Refactorings**
  - Cleans up session state and restores file status.
  - Handles and logs cleanup failures if they occur.

The test cases for this module can be found here.

4.4.7 Wipe Workspace Cache Command

Goal: This command allows users to reset all cached smell data and file statuses for the current workspace. The tests ensure correct user confirmation, safe cache clearing, and accurate feedback messaging across all user interaction scenarios.

This module's functionality is essential for maintaining control over stale or outdated results, especially in long development sessions. The tests verify proper UI messaging, cancellation handling, and full cache cleanup when confirmed.

Target requirement(s): FR10, OER-IAS5 [6]

- **Confirmation Prompt**
  - A warning dialog is shown before clearing the workspace cache.
  - The dialog must include a modal and an explicit "Confirm" action.
- **Cache Clearing and UI Refresh**
  - If the user confirms, all cached smells are deleted.
  - The SmellsView is refreshed and statuses are reset.
  - A success message is shown after completion.
- **Cancellation and Dismissal Handling**
  - If the user cancels or dismisses the dialog, no cache operations are performed.
  - A message confirms that the operation was cancelled.
- **Non-Confirm Input Handling**
  - If the user clicks any button other than "Confirm", the operation is treated as cancelled.
- **Message Validity**
  - Success messages are shown only after the cache is successfully cleared.
  - Cancel messages are mutually exclusive with success messages.

The test cases for this module can be found here.

4.4.8 Workspace Modified Listener

Goal: This listener monitors Python file changes and deletions within the configured workspace. Its responsibilities include invalidating outdated cache entries, updating UI statuses, and auto-triggering smell detection if "smell linting" is enabled. It also ensures resource cleanup upon disposal.

The following tests verify correct event handling on file saves, edits, and deletions. They also validate configuration checks, error handling, and the listener's lifecycle management.

Target requirement(s): FR10, FR16, OER-IAS5 [6]

- **Initialization Behavior**
  - Skips setup if no workspace path is configured.
  - Initializes file watcher for '.py' files when path is configured.
- **Handling File Changes**
  - Clears smell cache and marks file as outdated if it exists in the cache.
  - Skips files that are not in the cache.
  - Logs error messages for cache invalidation failures.
- **Handling File Deletions**
  - Clears cache and removes file entry from view if file had cached smells.
  - Skips deletion logic if file was not cached.
  - Logs error if cache clearing fails.
- **Smell Detection on Save**
  - Triggers smell detection when a Python file is saved and smell linting is enabled.
  - Ignores non-Python files.
- **Disposal**
  - Disposes of both the file watcher and save listener properly.
  - Logs disposal confirmation.

The test cases for this module can be found here.

4.4.9 Backend Service Communication

Goal: These backend API functions interact with the server for health checks, logging setup, smell detection, and smell refactoring. They must handle network issues gracefully, validate required inputs, and ensure proper feedback is logged in the output console.

The following tests validate success and failure scenarios for each backend endpoint, including malformed requests, network errors, and missing workspace paths. They also verify correct request payloads and logging output.

Target requirement(s): FR2, FR3, FR6, FR10, OER-IAS2 [6]

- **Server Health Check**
  - Sets status to `UP` if '/health' responds successfully.
  - Sets status to `DOWN` on error or network failure, and logs the issue.
- **Logging Initialization**
  - Sends proper payload to '/logs/init'.
  - Returns success on valid response.
  - Handles server or network errors with log messages and fallback.
- **Smell Detection**
  - Sends file path and smell configuration to '/smells'.
  - Parses and returns detected smells on success.
  - Throws clear errors and logs backend-provided details on failure.
- **Refactor Single Smell**
  - Sends smell and workspace path to '/refactor'.
  - Throws error when workspace path is missing.
  - Handles and logs both successful and failed backend responses.
- **Refactor by Smell Type**
  - Sends smell type and workspace path to '/refactor-by-type'.
  - Validates required fields and logs backend failures.

The test cases for this module can be found here.

4.4.10 File Highlighter

Goal: The file highlighter decorates code editor lines based on detected smells using user-defined styling preferences (underline, border, flashlight, etc.). This module ensures that the VS Code UI reflects current cache data and user configuration, applying or clearing decorations dynamically.

The tests verify correct singleton instantiation, dynamic decoration updates, conditional smell filtering, and visual style application across multiple configurations and editor scenarios.

Target requirement(s): FR8, FR13, OER-IAS1 [6]

- **Singleton Instantiation**
  - Ensures only one instance of the highlighter is created.
- **Highlight Update Triggers**
  - Calls `highlightSmells` only for visible Python editors.
  - Filters files correctly in `updateHighlightsForFile`.
- **Highlight Rendering**
  - Applies decorations for all enabled and valid smells.
  - Skips rendering if no data is cached or smell is disabled.
  - Ignores invalid line numbers.
- **Custom Decoration Styles**
  - Supports underline, flashlight, and border-arrow styles.
  - Defaults to underline for unknown style keys.
- **Decoration Disposal**
  - `resetHighlights()` clears all decorations and disposes them properly.

The test cases for this module can be found here.

4.4.11 Hover Manager

Goal: The hover manager displays contextual smell information when the user hovers over affected lines in Python files. It enriches the user experience by providing inline guidance and actionable links to trigger refactorings.

The tests verify correct registration of the hover provider, graceful degradation when conditions are unmet, and correct Markdown formatting and behavior for smells occurring on the hovered line.

Target requirement(s): FR13, FR16, OER-IAS1 [6]

- **Provider Registration**
  - Registers hover provider for `python` language files.
  - Adds registration to extension subscriptions.
- **Hover Preconditions**
  - Returns nothing for non-Python files.
  - Returns nothing when no smells are cached.
  - Returns nothing if no smells occur on the hovered line.
- **Hover Display Logic**
  - Creates and returns a `Hover` object with formatted message.
  - Displays smell description and a clickable refactor command.
  - Escapes special characters in Markdown to prevent formatting issues.

The test cases for this module can be found here.

4.4.12 Line Selection Manager

Goal: The line selection manager adds inline decorations that summarize code smells present on the user's current line selection. It enhances feedback clarity while ensuring only one comment is shown at a time.

The tests ensure that decoration is only applied to valid selections with associated smells, that it updates correctly when switching lines or clearing cache, and that decoration content reflects smell counts accurately.

Target requirement(s): FR13, FR16, OER-IAS1 [6]

- **Initialization**
  - Registers a callback on smell updates.
  - Initializes internal state to null decoration and null last line.
- **Comment Rendering**
  - Skips rendering when no editor is active or selection is multiline.
  - Skips when no smells are cached or no smells match the selected line.
  - Skips if user selects the same line twice.
  - Adds a comment with the smell type if one smell is present.
  - Adds a comment with count when multiple smells exist.
  - Decorations appear as inline text after the selected line.
- **Comment Removal**

- Removes previous decorations before applying a new one.
- Removes comment when cache is cleared for the current file or for all files.
- Skips removal if unrelated file's cache is cleared.

The test cases for this module can be found here.

4.4.13 Cache Initialization

Goal: This module restores file statuses from previously cached analysis results when the extension is activated. It removes entries for deleted or out-of-scope files and updates the UI accordingly.

The tests verify that cache entries outside the workspace or pointing to non-existent files are removed, and that remaining entries are properly restored into the UI. It also validates summary log reporting and gracefully handles missing workspace configuration.

Target requirement(s): FR10, OER-CACHE1 [6]

- **No Workspace Configured**
  - Skips initialization and logs a warning if no workspace path is set.
- **Removing Invalid Cache Entries**
  - Removes entries for files outside the configured workspace.
  - Removes entries for files that are no longer present on disk.
- **Restoring Valid Cache**
  - Files with smells get status `passed` and corresponding smells are injected.
  - Clean files are marked with status `no_issues`.
- **Summary Logging**
  - Logs how many files were restored, how many had smells, and how many were removed from the cache.

The test cases for this module can be found here.

4.4.14 Smells Data Management

Goal: This module provides utilities for reading, parsing, and retrieving code smell configurations. It handles loading/saving JSON configurations, filtering enabled smells, and resolving smell metadata for hover/tooltips.

The tests validate the integrity and correctness of the config loading and writing process, proper filtering of enabled smells, and lookup logic for acronyms, names, and descriptions based on message IDs.

Target requirement(s): FR4, FR5, OER-CONFIG1 [6]

- **loadSmells**
  - Successfully loads a smells configuration from disk.
  - Displays an error message if the file is missing or malformed.
- **saveSmells**
  - Writes updated configuration to disk.
  - Displays an error message if the save operation fails.
- **getFilterSmells**
  - Returns the full dictionary of loaded smells.
- **getEnabledSmells**
  - Returns only smells marked as enabled.
  - Includes parsed analyzer options with correct types.
- **Metadata Resolvers**
  - `getAcronymByMessageId` resolves acronyms.
  - `getNameByMessageId` resolves full names.
  - `getDescriptionByMessageId` resolves descriptions.

The test cases for this module can be found here.

4.4.15 Tracked Diff Editors

Goal: This module maintains a registry of active diff editors created during refactoring sessions. It provides utilities to register, query, and programmatically close diff tabs within the VS Code environment.

The tests validate the correct registration and tracking of diff editors, accurate identification of tracked editors via URI comparison, and complete cleanup of tabs and memory state after closure operations.

Target requirement(s): FR12, OER-IAS2 [6]

- **registerDiffEditor**
  - Adds a URI pair to the tracked diff editors set.
  - Supports multiple pairs without interference.
- **isTrackedDiffEditor**
  - Returns true only for previously registered URI pairs.
  - Fails gracefully for unregistered or mismatched pairs.

- Is case-sensitive when comparing URI strings.

- **closeAllTrackedDiffEditors**
  - Closes all diff tabs currently open in the workspace if they match tracked URIs.
  - Skips irrelevant tabs or malformed inputs.
  - Clears the internal tracked set after execution.

The test cases for this module can be found here.

4.4.16 Refactor Action Buttons

Goal: These UI buttons provide users with the ability to accept or reject a proposed refactoring. They must be initialized correctly, displayed when a refactoring session is active, and hidden when the session ends. The buttons are tied to a VS Code context key used to control view behavior.

The tests verify that the buttons are properly initialized and registered with the extension context, appear or disappear based on user interaction, and correctly set or reset the `refactoringInProgress` context variable.

Target requirement(s): FR12 [6]

- **Button Visibility**
  - The accept and reject buttons are shown when a refactoring session begins.
  - The buttons are hidden when the session ends or is cancelled.

- **Context Key Updates**
  - `refactoringInProgress` is set to `true` when buttons are shown.
  - `refactoringInProgress` is set to `false` when buttons are hidden.

- **Subscription Registration**
  - Buttons are registered to the extension context upon initialization.

The test cases for this module can be found here.

References

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

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

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

[4] 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.

[5] 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.

[6] 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.

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

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

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

[10] G. van Rossum, B. Warsaw, and A. Coghlan. Pep 8 – style guide for python code. https://peps.python.org/pep-0008/. Accessed: 2024-11-03.

Appendices

A Appendix

A.1 Symbolic Parameters

Not applicable at the moment.

A.2 Usability Survey Questions

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

B Reflection

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

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

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

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

1. What went well while writing this deliverable?

2. What pain points did you experience during this deliverable, and how did you resolve them?

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

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

Mya Hussain

- *What went well while writing this deliverable?*

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

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

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

Sevhena Walker

- *What went well while writing this deliverable?*

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

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

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

Nivetha Kuruparan

- *What went well while writing this deliverable?*

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

  Diving into the Design Verification Plan in section 3.3 was another highlight for me. It helped me get a better grasp of the requirements from the SRS. I felt more confident knowing we had a solid verification approach that covered all the bases, including

functional and non-functional requirements. The discussions about incorporating static verification techniques and the importance of regular peer reviews were eye-opening and really enhanced our strategy for maintaining code quality.

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

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

Ayushi Amin

- *What went well while writing this deliverable?*

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

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

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

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

  Another tough spot was figuring out how to handle usability and performance testing in a way that doesn't feel overly theoretical. Since we're not at the implementation stage, it's hard to gauge what users will really need. To work through this, I focused

on drawing from what we know about our end-users and aligning our plan with the goals outlined in the SRS. Keeping that user-centered perspective helped ground the plan, making it feel more actionable even at this early stage.

Tanveer Brar

- *What went well while writing this deliverable?*

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

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

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

Group Reflection

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

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

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

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

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

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

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

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

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

  Sevhena Walker (Lead Tester)

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

Mya Hussain (Functional Requirements Tester)

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

Ayushi Amin (Integration Tester)

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

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

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

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

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

    * Online UX/UI Design Courses: Enroll in courses that focus on usability principles and user experience design.
  - **Preferred Approach:** User Feedback Analysis
  - **Reason:** This approach provides real-world insights into how the product is perceived and used, making adjustments more relevant.

Istvan David (Supervisor)

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