Verification and Validation Report: Software Engineering

Team 4, EcoOptimizers

Nivetha Kuruparan
Sevhena Walker
Tanveer Brar
Mya Hussain
Ayushi Amin

March 11, 2025

Revision History

Date	Version	Notes
March 8th, 2025	0.0	Started VnV Report

Symbols, Abbreviations and Acronyms

symbol	description
T	Test
TC	Test Case
VSCode	Visual Studio Code

Contents

List of Tables

List of Figures

This Verification and Validation (V&V) report outlines the testing process used to ensure the accuracy, reliability, and performance of our system. It details our verification approach, test cases, and validation results, demonstrating that the system meets its requirements and functions as intended. Key findings and resolutions are also discussed.

1 Functional Requirements Evaluation

1.1 Code Input Acceptance Tests

1. **test-FR-1A Valid Python File Acceptance**

   The **valid Python file acceptance test** ensures that the system correctly processes a syntactically valid Python file without errors. A correctly formatted Python file was provided as input, and the expected result was that the system should accept the file without issue. The **actual result** confirmed that the system successfully processed the valid file without generating any errors.

2. **test-FR-1A-2 Feedback for Python File with Bad Syntax**

   This test verifies that the system correctly handles Python files containing deliberate syntax errors. A Python file with syntax errors was fed into the system, and the expected result was that the system should reject the file and provide an appropriate error message indicating the syntax issue. The **actual result** confirmed that the system correctly identified the syntax errors and displayed the expected error message.

3. **test-FR-1A-3 Feedback for Non-Python File**

   The **non-Python file test** ensures that the system correctly rejects unsupported file types and provides clear feedback. A document file (`document.txt`) and a script with an incorrect file extension (`script.js`) were tested. The expected result was that the system should reject the files and return an error message indicating that the file format is not supported. The **actual result** confirmed that the system correctly flagged the non-Python files and provided the appropriate error message.

1.2 Code Smell Detection and Refactoring Suggestion (RS) Tests

1. **test-FR-2 Code Smell Detection and Refactoring Suggestion**

   The **code smell detection and refactoring tests** validate the system's ability to identify and refactor specific code smells that impact energy efficiency. These tests ensure compliance with **functional requirement FR2** and were conducted through unit testing.

   The tester provided Python files containing common code smells such as **long parameter lists, repeated function calls, and inefficient string concatenation**. The **expected result** was that the system would correctly detect these smells and suggest appropriate refactoring strategies. The **actual result** confirmed that the sys-

tem successfully identified all tested code smells, displayed warnings, and provided optimization suggestions. More details can be found in the unit tests.

1.3 Output Validation Tests

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

   The **output validation test** ensures that refactored Python code remains syntactically correct and compliant with Python standards. This validation is crucial for maintaining **functional requirement FR3**, as it confirms that the refactored code behaves identically to the original but with improved efficiency.

   A Python file with detected code smells was refactored, and the expected result was that the optimized code should pass a syntax check and retain its original functionality. The **actual result** confirmed that the refactored code was valid, passed linting checks, and maintained correctness.

1.4 Tests for Reporting Functionality

The reporting functionality of the tool is a critical feature that provides comprehensive insights into the refactoring process, including detected code smells, applied refactorings, energy consumption measurements, and test results. These tests ensure that the reporting feature operates correctly and delivers accurate, well-structured information as specified in **functional requirement FR9**.

At this stage, the reporting functionality is still under development, and testing has not yet been conducted. The tests outlined below will be performed in **Revision 1** once the reporting feature is fully implemented.

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

   This test ensures that the tool generates a comprehensive report that includes all necessary information required by **FR9**. The system should produce a structured summary of the refactoring process, displaying detected code smells, applied refactorings, and energy consumption metrics.

   Planned Test Execution: After refactoring, the tool will invoke the report generation feature, and a user will validate that the output meets the structure and content specifications.

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

   This test will verify that the report correctly includes details on detected code smells and refactorings, ensuring compliance with **FR9**.

   Planned Test Execution: The tool will generate a report, and its contents will be compared with the detected code smells and refactorings to confirm accuracy.

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

This test will validate that the reporting feature correctly includes energy consumption measurements before and after refactoring, aligning with **FR9**.

Planned Test Execution: A user will analyze the energy consumption metrics in the generated report to ensure they accurately reflect the measurements taken during the refactoring process.

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

   This test will ensure that the reporting functionality accurately reflects the results of the test suite, summarizing test pass/fail outcomes after refactoring.

   Planned Test Execution: The tool will generate a report, and validation will be conducted to confirm that it includes a summary of test results matching the actual execution outcomes.

1.5 Documentation Availability Tests

The following tests will ensure that the necessary documentation is available as per **FR10**. Since documentation is still under development, these tests have not yet been conducted and will be included in **Revision 1**.

1. **test-FR-DA-1 Test for Documentation Availability**

   This test verifies that the system provides proper documentation covering installation, usage, and troubleshooting.

   Planned Test Execution: Review the documentation for completeness, clarity, and accuracy, ensuring it meets **FR10**.

1.6 IDE Extension Tests

The following tests ensure that users can integrate the tool as a VS Code extension in compliance with **FR11**. Local testing has been conducted successfully, confirming the extension's ability to function within the development environment. Once all features are implemented, the extension will be packaged and tested in a deployed environment.

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

   This test ensures that the refactoring tool extension can be installed from the Visual Studio Marketplace.

   Test Execution: The extension was installed locally, and its presence in the Extensions View was confirmed.

   Future Testing: Once all features are implemented, the extension will be zipped, packaged, and tested as a published extension.

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

   This test validates that the extension functions correctly within the development environment, detecting code smells and suggesting refactorings.

Test Execution: Local tests confirmed that activating the extension successfully detects code smells and applies refactorings.

Future Testing: Once the extension is packaged, additional tests will be conducted to confirm functionality in a deployed environment.

2 Nonfunctional Requirements Evaluation

2.1 Usability

Key Findings

- The extension demonstrated strong functionality in detecting code smells and providing refactoring suggestions.
- Participants appreciated the **preview feature** and **energy savings feedback**.
- Major usability issues included **sidebar visibility**, **refactoring speed**, and **UI clarity**.

Methodology

The usability test involved 5 student developers familiar with VSCode but with no prior experience using the extension. Participants performed tasks such as detecting code smells, refactoring single and multi-file smells, and customizing settings. Metrics included task completion rate, error rate, and user satisfaction scores. Additional qualitative data was collected using surveys that gathered background information of the participants as well as their opinions post testing (9.2).

Results

The following is an overview of the most significant task that the test participants performed. Information on the tasks themselves can be found in the Appendix (9.2).

Quantitative Results

- **Task Completion Rate:**
  - **Task 1-3 (Smell Detection):** 100% success rate.
  - **Task 4 (Initiate Refactoring):** 100% success rate.
  - **Task 6 (Multi-File Refactoring):** 60% success rate (participants struggled with identifying clickable file names).
  - **Task 7 (Smell Settings):** 100% success rate.
- **User Satisfaction:**

– Confidence in Using the Tool: **4.2/5**.
– Satisfaction with UI Design: **4.0/5**.
– Trust in Refactoring Suggestions: **4.5/5**.

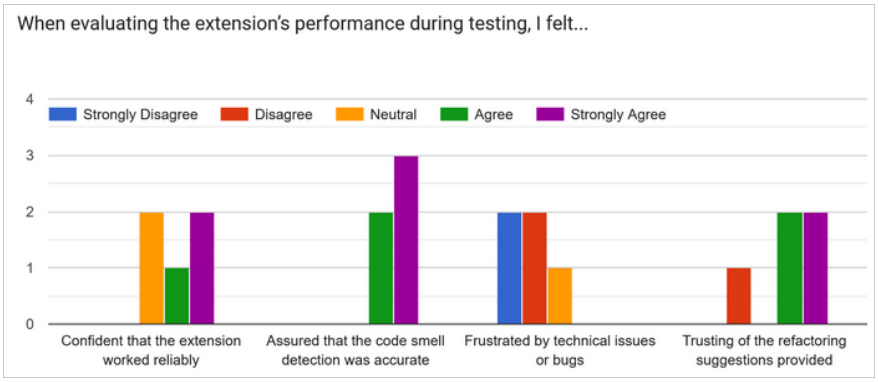


Figure 1: User Satisfaction Survey Data

Qualitative Results

Participants found the code smell detection intuitive and accurate, and they appreciated the preview feature and Accept/Reject buttons. However, they struggled with sidebar visibility, refactoring speed, and UI clarity. Hover descriptions were overwhelming, and some elements (e.g., "(6/3)") were unclear.

Discussion

The usability test revealed that the extension performs well in detecting code smells and providing refactoring suggestions. Participants appreciated the energy savings feedback but requested clearer explanations of how refactoring improves energy efficiency. The sidebar and refactoring process were identified as major pain points, requiring immediate attention.

The extension met its core functionality objectives but fell short in UI clarity and performance reliability. Participants expressed interest in using the extension in the future, provided the identified issues are addressed. The test highlighted the need for better onboarding, clearer documentation, and performance optimizations to enhance user satisfaction and adoption.

Feedback and Implementation Plan

The following table summarizes participant feedback and whether the suggested changes will be implemented:

Feedback	Implementation Decision	Reason
Relocate the sidebar or change its colour for better visibility.	Partial	The relocation of the sidebar is not something that is in scope during the development period.
Make Accept/Reject buttons more prominent and visually distinct.	Yes	High user frustration.
Allow users to customize colours for different types of smells.	Yes	Enhances user experience.
Optimize the refactoring process to reduce wait times.	No	This is a time intensive ask that is not in scope.
Add progress bars or loading messages to manage user expectations.	Yes	Additional messages will be added to the UI.
Provide step-by-step instructions and a tutorial for new users.	Yes	This was already planned and will be implemented for revision 1.
Simplify hover descriptions and provide examples or links to documentation.	Yes	The hover content will be improved for revision 1.
Explain how refactoring saves energy, possibly with visualizations.	Partial	No visualizations will be added, but better explanation of smells will be provided.

Table 1: Participant Feedback and Implementation Decisions

2.2 Performance

This testing benchmarks the performance of ecooptimizer across files of varying sizes (250, 1000, and 3000 lines). The data includes detection times, refactoring times for specific smells, and energy measurement times. The goal is to identify scalability patterns, performance bottlenecks, and opportunities for optimization.

Related Performance Requirement: PR-1

The test cases for this module can be found here

This script benchmarks the following components:

1. **Detection/Analyzer Runtime** (via `AnalyzerController.run_analysis`)

2. **Refactoring Runtime** (via `RefactorerController.run_refactorer`)

3. **Energy Measurement Time** (via `CodeCarbonEnergyMeter.measure_energy`)

For each detected smell (grouped by smell type), refactoring is run 10 times to compute average times.

The following is for your reference:

Type of Smell	Code	Smell Name
Pylint	R0913	Long Parameter List
Pylint	R6301	No Self Use
Pylint	R1729	Use a Generator
Custom	LMC001	Long Message Chain
Custom	UVA001	Unused Variable or Attribute
Custom	LEC001	Long Element Chain
Custom	LLE001	Long Lambda Expression
Custom	SCL001	String Concatenation in Loop
Custom	CRC001	Cache Repeated Calls

1. Detection Time vs File Size

Figure 2: Detection Time vs File Size

What: Linear plot showing code smell detection time growth with file size

Why: Understand scalability of detection mechanism

The detection time grows non-linearly with file size, suggesting a potential $O(n^2)$ complexity. For a 250-line file, detection takes 0.38 seconds, while a 1000-line file takes 0.90 seconds (a 2.4× increase). At 3000 lines, the detection time jumps to 2.58 seconds (a 2.9× increase from 1000 lines). This indicates that the detection algorithm scales poorly for larger files, which could become problematic for very large codebases. However, the absolute times remain reasonable, with detection completing in under 3 seconds even for 3000-line files making this not a current critical bottleneck.

2. Refactoring Times by Smell Type (Log Scale)

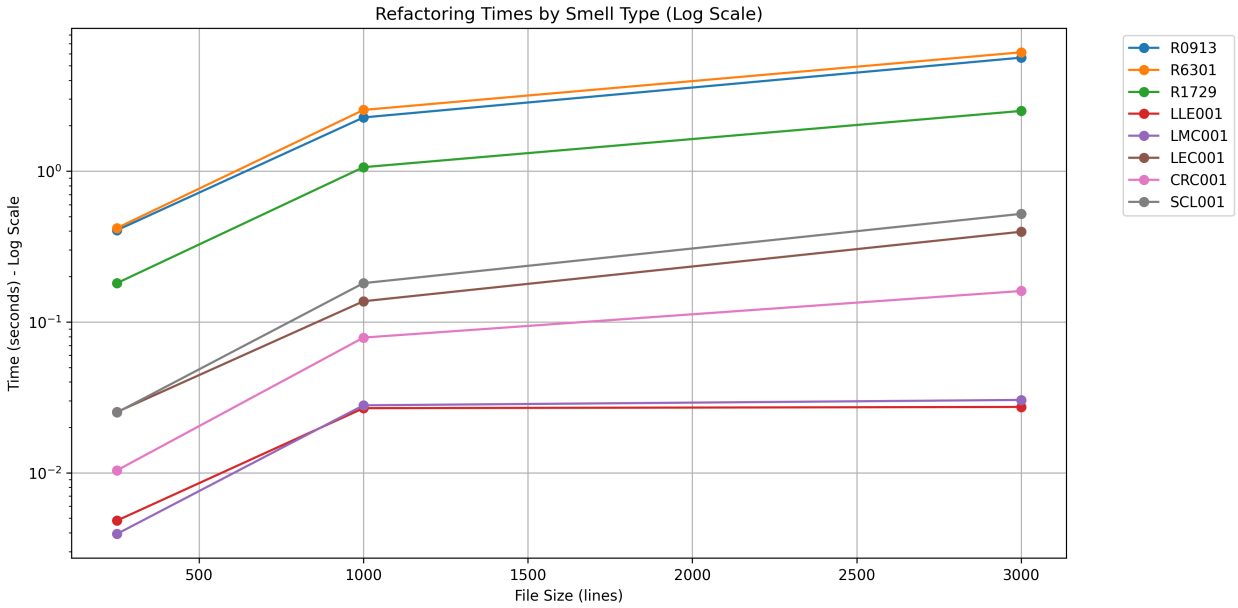


Figure 3: Refactoring Times by Smell Type (Log Scale)

What: Logarithmic plot of refactoring times per smell across file sizes

Why: Identify most expensive refactorings and scalability patterns

The logarithmic plot reveals a clear hierarchy of refactoring costs. The most expensive smells are `R6301` and `R0913`, which take 6.13 seconds and 5.65 seconds, respectively, for a 3000-line file. These smells show exponential growth, with `R6301` increasing by 14.6× from 250 to 3000 lines. In contrast, low-cost smells like `LLE001` and `LMC001` remain consistently fast (0.03 seconds) across all file sizes. This suggests that optimizing `R6301` and `R0913` should be a priority, as they dominate the refactoring time for larger files.

3. Refactoring Times Heatmap

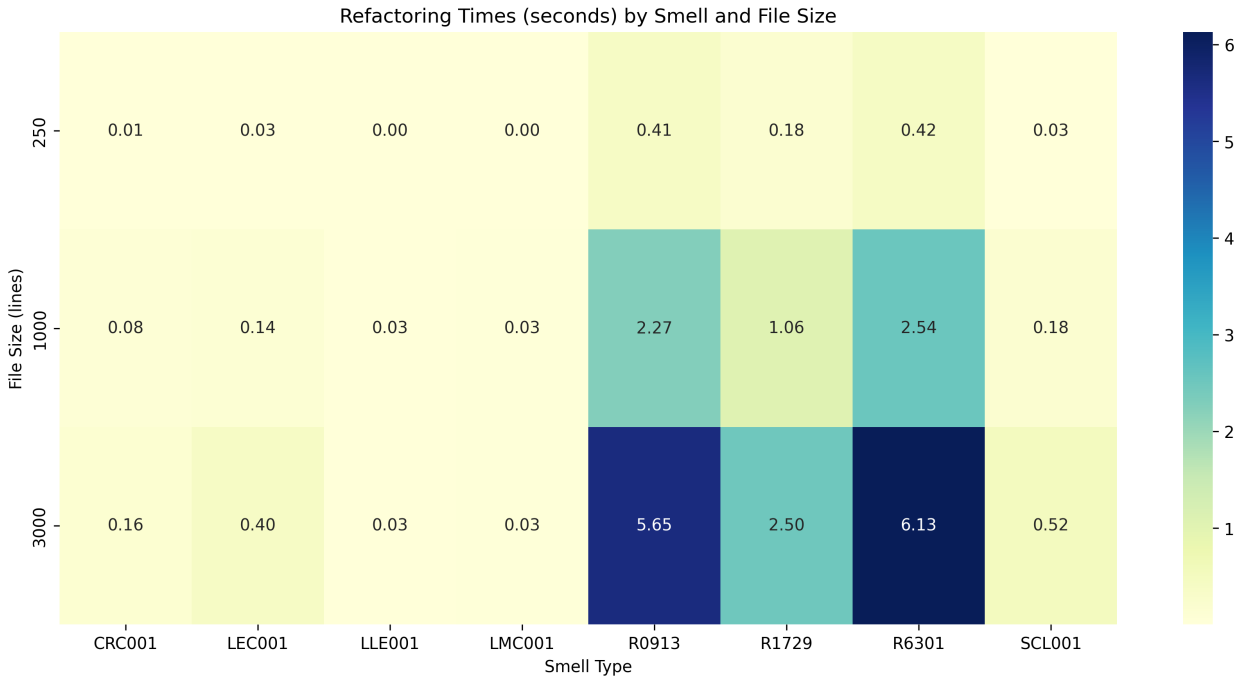


Figure 4: Refactoring Times Heatmap

What: Color-coded matrix of refactoring times by smell/file size

Why: Quick visual identification of hot spots

The heatmap provides a quick visual summary of refactoring times across smells and file sizes. The darkest cells correspond to `R6301` and `R0913` at 3000 lines, confirming their status as the most expensive operations. In contrast, `LLE001` and `LMC001` remain light-colored across all sizes, indicating consistently low costs. The heatmap also highlights the dramatic variation in refactoring times: at 3000 lines, the fastest smell (`LLE001`) is 200× faster than the slowest (`R6301`).

4. Energy Measurement Times Distribution

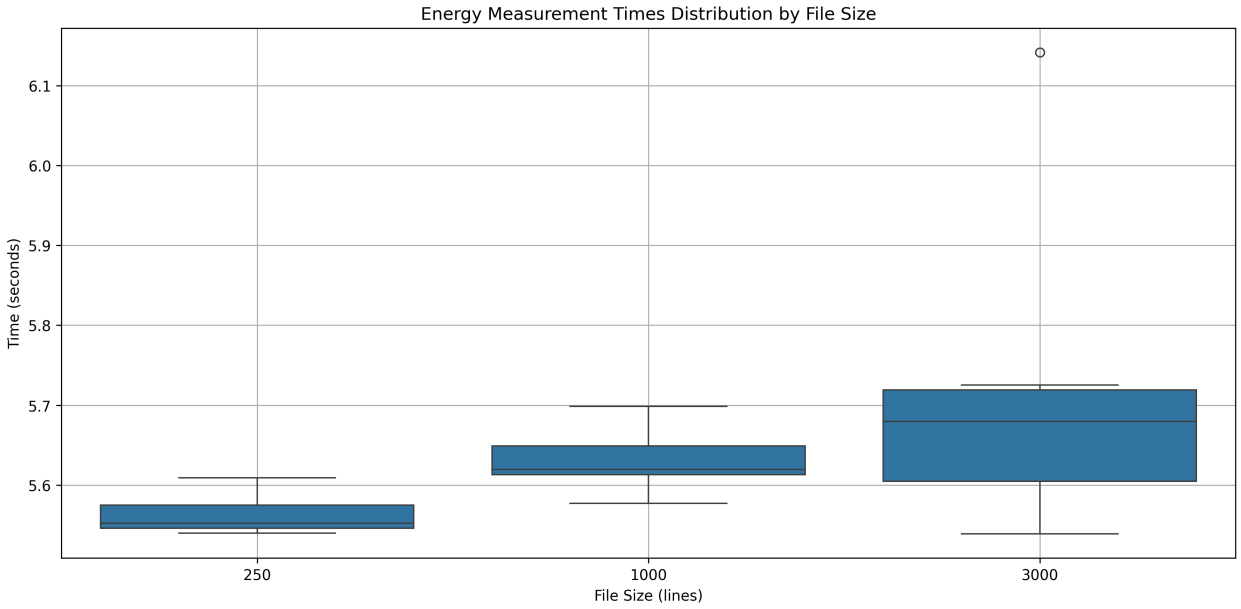


Figure 5: Energy Measurement Times Distribution

What: Box plot of energy measurement durations

Why: Verify measurement consistency across operations

Energy measurement times are remarkably consistent, ranging from 5.54 to 6.14 seconds across all operations and file sizes. The box plot shows no significant variation with file size, suggesting that energy measurement is operation-specific rather than dependent on the size of the file. This stability could indicate that the energy measurement process has a fixed overhead, which could simplify efforts in the future if we were to create our own energy measurement module.

5. Comparative Refactoring Times per File Size

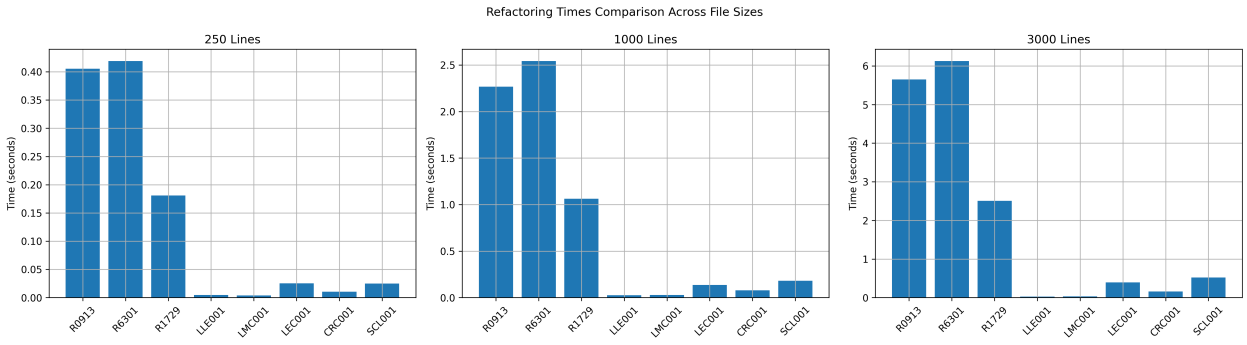


Figure 6: Comparative Refactoring Times per File Size

What: Side-by-side bar charts per file size

Why: Direct comparison of refactoring costs at different scales

The side-by-side bar charts reveal consistent dominance patterns across file sizes. `R6301` and `R0913` are always the top two most expensive smells, while `LLE001` and `LMC001` remain the cheapest. Notably, the relative cost difference between the most and least expensive smells increases with file size: at 250 lines, the ratio is 100:1, but at 3000 lines, it grows to 200:1. This suggests that the scalability of refactoring operations varies significantly by smell type.

6. Energy vs Refactoring Time Correlation

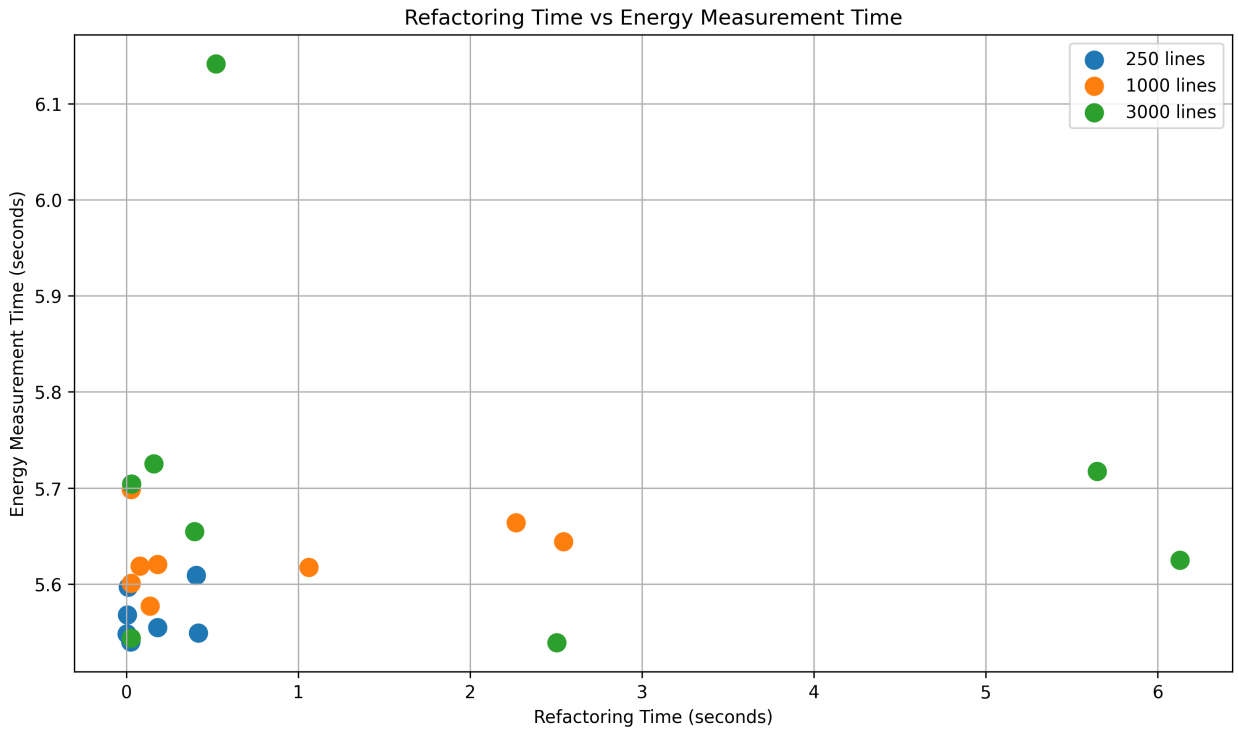


Figure 7: Energy vs Refactoring Time Correlation

What: Scatter plot comparing refactoring and energy measurement times

Why: Identify potential relationships between effort and energy impact

The scatter plot shows no clear correlation between refactoring time and energy measurement time. Fast refactorings like `LLE001` and slow refactorings like `R6301` both result in energy measurement times clustered between 5.5 and 6.1 seconds. This makes perfect sense as the refactoring operations and energy measurement are disjoint functionalities in the code.

Key Insights and Recommendations

- **Bottleneck Identification:** The smells `R6301` and `R0913` are the primary bottlenecks, consuming over 50% of the total refactoring time for 3000-line files. Optimizing these operations should be a top priority.
- **Scalability Concerns:** Both detection and refactoring times scale poorly with file size, suggesting $O(n^2)$ complexity. This could become problematic for very large codebases.

- **Low-Hanging Fruit:** Smells like `LLE001` and `LMC001` are consistently fast to refactor, making them ideal candidates for early refactoring efforts.
- **Energy Measurement Stability:** Energy measurement times seem consistent across operations and file sizes, indicating a fixed overhead. This simplifies efforts to correlate refactoring with energy savings.
- **Disproportionate Costs:** The cost difference between the most and least expensive smells grows with file size, highlighting the need for targeted optimization.

The analysis reveals significant scalability challenges for both detection and refactoring, particularly for smells like `R6301` and `R0913`. While energy measurement times are stable, their lack of correlation with refactoring time suggests that additional metrics may be needed to accurately assess energy savings. Future work should focus on optimizing high-cost operations and improving the scalability of the detection algorithm.

2.3 Maintainability and Support

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

   To validate the extensibility of our tool, we structured the codebase using a modular design, where new code smell detection and refactoring functions can be easily added as separate components. In simpler terms, each refactoring and each custom detection is placed in its own file. A code walkthrough confirmed that existing modules remain unaffected when adding new detection logic. We successfully integrated a sample code smell and its corresponding refactoring method with minimal changes, ensuring that the new function was accessible through the main interface. This demonstrated that our architecture supports future expansions without disrupting core functionality.

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

   We conducted a static analysis and documentation walkthrough to assess the maintainability of our codebase. The code was reviewed for modular organization, clear separation of concerns, and adherence to coding standards. All modules were correctly seperated and organzied. Documentation will be updated to include detailed descriptions of functions and configuration files, ensuring clarity for future developers. Code comments were refined to enhance readability, and function naming conventions were standardized for consistency. These efforts ensured that the tool remains adaptable to new Python versions and evolving best practices.

3. **test-MS-3: Easy rollback of updates in case of errors**

   Once releases are made, each release will be properly tagged and versioned to ensure smooth rollbacks through version control. This will all be handled with Git. done, but this approach guarantees that users will be able to revert to a previous stable version if needed, maintaining system integrity and minimizing disruptions.

2.4 Look and Feel

1. **test-LF-1 Side-by-Side Code Comparison in IDE Plugin**

   The side-by-side code comparison feature in the IDE plugin was tested manually to verify that users can clearly view the original and refactored code within the VS Code interface. The test followed the procedure outlined in Test-LF-1, which specifies that upon initiating a refactoring operation, the plugin should display the original and modified versions of the code in parallel, allowing users to compare changes effectively.

   The tester performed the test dynamically by opening a sample code file within the plugin and applying various refactoring operations across all detected code smells. The expected result was that the IDE plugin would correctly display the two versions side by side, with clear options for users to accept or reject each change. The actual result confirmed that the functionality operates as expected: refactored code was displayed adjacent to the original code, ensuring an intuitive comparison process. The tester was also able to interact with the accept/reject buttons, verifying their usability and correctness.

   A screenshot of the successful test execution is provided in Figure 8, illustrating the side-by-side code comparison functionality within the IDE plugin.

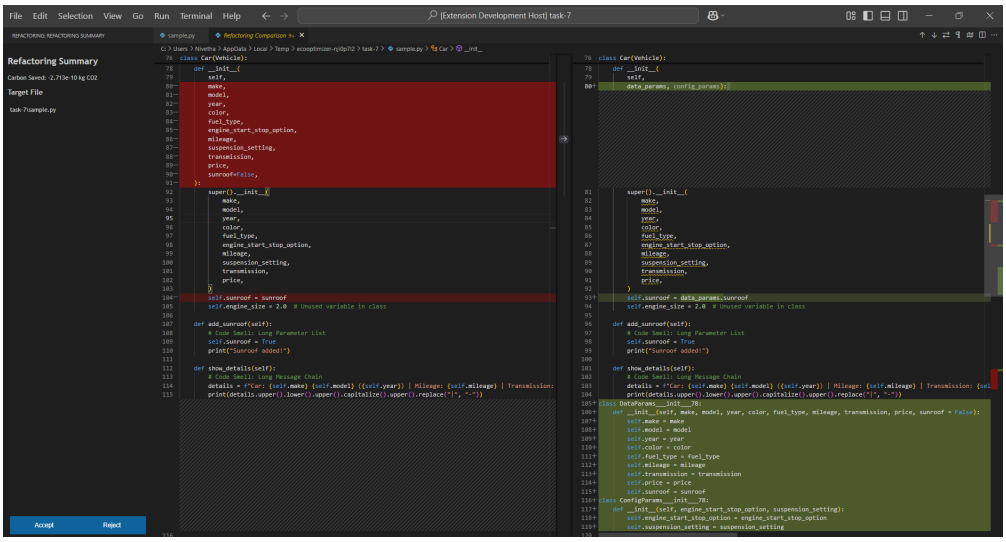


Figure 8: Side-by-Side Code Comparison in VS Code Plugin

2. **test-LF-2 Theme Adaptation in VS Code**

   The theme adaptation feature in the IDE plugin was tested manually to confirm that the plugin correctly adjusts to VS Code's light and dark themes without requiring manual configuration. The tester performed the test by opening the plugin in both themes and switching between them using VS Code's settings.

   The expected result was that the plugin's interface should automatically adjust when the theme is changed. The actual result confirmed that the plugin seamlessly transitioned between light and dark themes while maintaining a consistent interface. The

images in Figures 9 and 10 illustrate the side-by-side refactoring panel in both light mode and dark mode.

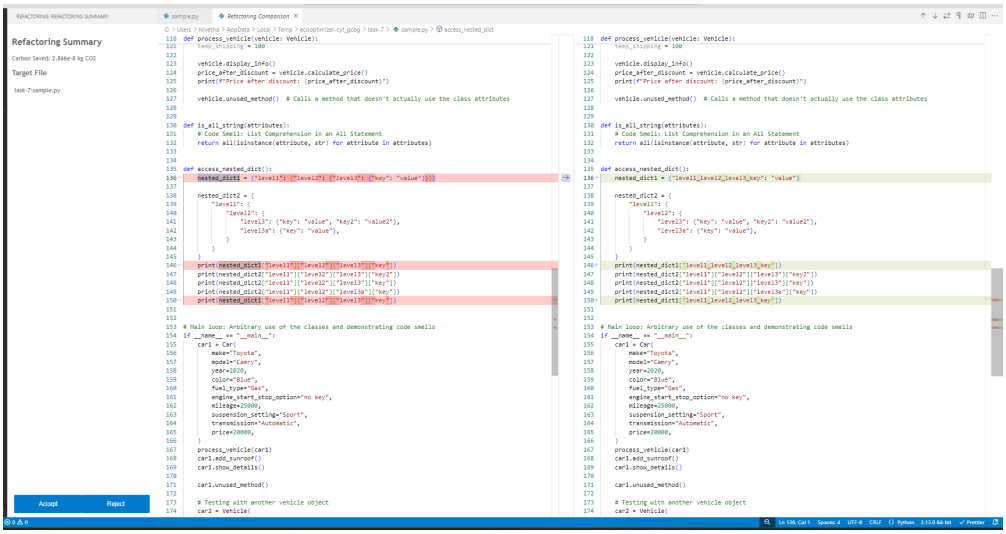


Figure 9: Side-by-Side Refactoring Panel in Light Mode

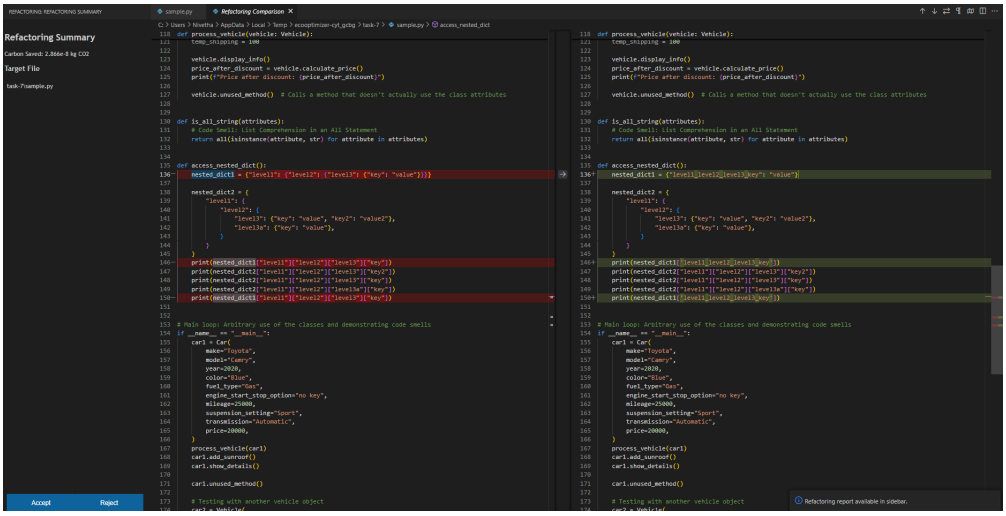


Figure 10: Side-by-Side Refactoring Panel in Dark Mode

3. **test-LF-3 Design Acceptance**

   The design acceptance test was conducted as part of the usability testing session, where developers and testers interacted with the plugin and provided feedback. This test evaluated user experience, ease of navigation, and overall satisfaction with the plugin's interface.

   The expected result was that users would be able to interact with the plugin smoothly and provide structured feedback. The actual result confirmed that users were able to navigate and use the plugin effectively. The feedback collected during this session was

used to assess the overall usability of the plugin. More details regarding this evaluation can be found in the Usability Testing section.

2.5 Operational & Environmental

test-OPE-1 will be tested once the extension is officially launched.

test-OPE-2 tests a feature that is yet to be implemented.

test-OPE-3 will be tested once the python package is published.

2.6 Security

test-SRT-1: Audit Logs for Refactoring Processes

We conducted a combination of code walkthroughs and static analysis of logging mechanisms to validate that the tool maintains a secure log of all refactoring processes, including pattern analysis, energy analysis, and report generation. The objective was to do so while covering the logging mechanisms for refactoring events, ensuring that logs are complete and immutable.

The development team reviewed the codebase to confirm that each refactoring event (pattern analysis, energy analysis, report generation) is logged with accurate timestamps and event description. Missing log entries and/or insufficient details were identified and added to the logging process.

Through this process, all major refactoring processes were correctly logged with accurate timestamps. Logs are stored locally on the user's device, ensuring that unauthorized modifications are prevented by restricting external access.

The team was able to confirm that the tool maintains a secure logging system for refactoring processes, with logs being tamper-resistant due to their local storage on user devices.

2.7 Compliance

1. **test-CPL-1: Compliance with PIPEDA and CASL**

   This process was applied to all processes related to data handling and user communication within the local API framework with the objective of assesing whether the tool's data handling and communication mechanisms align with PIPEDA and CASL requirements, ensuring that no personal information is stored, all processing is local, and communication practices meet regulatory standards.
   Through code review, the team confirmed that all data processing remains local and does not involve external storage. During this time, internal API functionality was also reviewed to ensure that user interactions are transient and not logged externally.

By going through the different workflows, the team verified that no personal data collection occurs, eliminating the need for explicit consent mechanisms.
As a result of this process, it was concluded that the tool does not store any user data. The tool also does not send unsolicited communications, aligning with CASL requirements.

2. **test-CPL-2: Compliance with ISO 9001 and SSADM Standards**

   This process evaluated development workflows, documentation practices, and adherence to structured methodologies with the object of assessing whether the tool's quality management and software development processes align with ISO 9001 standards for quality management and SSADM for structured software development.
   Through an unbiased approach, the team verified the presence of structured documentation, feedback mechanisms, and version tracking. It was also confirmed that a combination of unit testing, informal testing and iteration processes were applied during development. After code review, adherence to structured programming and modular design principles was also confirmed. Our goal was to take a third perspective check on whether these set of practices were applied to our development workflows. Development follows reasonable structured processes and also includes formal documentation of testing and quality assurance procedures. Version control system is present including change tracking and basic project management.

3 Comparison to Existing Implementation

Not applicable.

4 Unit Testing

The following section outlines the unit tests created for the python backend modules and the vscode extension.

4.1 API Endpoints

4.1.1 Smell Detection Endpoint

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC1	FR10, OER-IAS1	User requests to detect smells in a valid file.	Status code is 200. Response contains 2 smells.	All assertions pass.	Pass

Continued on next page

(Continued from previous page)

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC2	FR10, OER-IAS1	User requests to detect smells in a non-existent file.	Status code is 404. Error message indicates file not found.	All assertions pass.	Pass
TC3	FR10, OER-IAS1	Internal server error occurs during smell detection.	Status code is 500. Error message indicates internal server error.	All assertions pass.	Pass

Table 2: Smell Detection Endpoint Test Cases

4.1.2 Refactor Endpoint

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC4	FR10, OER-IAS1	User requests to refactor a valid source directory.	Status code is 200. Response contains refactored data and updated smells.	All assertions pass.	Pass
TC5	FR10, OER-IAS1	User requests to refactor a non-existent source directory.	Status code is 404. Error message indicates directory not found.	All assertions pass.	Pass
TC6	FR10, OER-IAS1	Energy is not saved after refactoring.	Status code is 400. Error message indicates energy was not saved.	All assertions pass.	Pass
TC7	FR10, OER-IAS1	Initial energy measurement fails.	Status code is 400. Error message indicates initial emissions could not be retrieved.	All assertions pass.	Pass

Continued on next page

(Continued from previous page)

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC8	FR10, OER-IAS1	Final energy measurement fails.	Status code is 400. Error message indicates final emissions could not be retrieved.	All assertions pass.	Pass
TC9	FR10, OER-IAS1	Unexpected error occurs during refactoring.	Status code is 400. Error message contains the exception details.	All assertions pass.	Pass

Table 3: Refactor Endpoint Test Cases

4.2 Analyzer Controller Module

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC10	FR2, FR5, PR-PAR3	Test detection of repeated function calls in AST-based analysis.	One repeated function call should be detected.	All assertions pass.	Pass
TC11	FR2, FR5, PR-PAR3	Test detection of repeated method calls on the same object instance.	One repeated method call should be detected.	All assertions pass.	Pass
TC12	FR2	Test that no code smells are detected in a clean file.	The system should return an empty list of smells.	All assertions pass.	Pass
TC13	FR2, PR-PAR2	Test filtering of smells by analysis method.	The function should return only smells matching the specified method (AST, Pylint, Astroid).	All assertions pass.	Pass

Continued on next page

(Continued from previous page)

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC14	FR2, PR-PAR2	Test generating custom analysis options for AST-based analysis.	The generated options should include callable detection functions.	All assertions pass.	Pass
TC15	FR2, FR5, PR-PAR3	Test correct logging of detected code smells.	Detected smells should be logged with correct details.	All assertions pass.	Pass
TC16	FR2, FR5	Test handling of an empty registry when filtering smells.	The function should return an empty dictionary.	All assertions pass.	Pass
TC17	FR2, PR-PAR2	Test that smells remain unchanged if no modifications occur.	The function should not modify existing smells if no changes are detected.	All assertions pass.	Pass

Table 4: Analyzer Controller Module Test Cases

4.3 CodeCarbon Measurement

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC18	PR-RFT1, FR6	Trigger CodeCarbon measurements with a valid file path.	CodeCarbon subprocess for the file should be invoked at least once. `EmissionsTracker.start` and `stop` API endpoints should be called. Success message "CodeCarbon measurement completed successfully." should be logged.	All assertions pass.	Pass
TC19	PR-RFT1	Trigger CodeCarbon function with a valid file path that causes a subprocess failure.	CodeCarbon subprocess run should still be invoked. `EmissionsTracker.start` and `stop` API endpoints should be called. An error message "Error executing file" should be logged. Returned emissions data should be `None` since the execution failed.	All assertions pass.	Pass
TC20	FR5, PR-SCR1	Results produced by CodeCarbon run are at a valid CSV file path and can be read.	Emissions data should be read successfully from the CSV file. The function should return the last row of emissions data.	All assertions pass.	Pass

Continued on next page

(Continued from previous page)

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC21	PR-RFT1, FR6	Results produced by CodeCarbon run are at a valid CSV file path but the file cannot be read.	An error message "Error reading file" should be logged. The function should return `None` because the file reading failed.	All assertions pass.	Pass
TC22	PR-RFT1, FR5	Given CSV Path for results produced by CodeCarbon does not have a file.	An error message "File `file path` does not exist." should be logged.The function should return `None` since the file does not exist.	All assertions pass.	Pass

Table 5: CodeCarbon Measurement Test Cases

4.4 Smell Analyzers

4.4.1 String Concatenation in Loop

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC23	FR2	Detects `+=` string concatenation inside a `for` loop.	One smell detected with target `result` and line 4.	All assertions pass.	Pass
TC24	FR2	Detects `<var = var + ...>` string concatenation inside a loop.	One smell detected with target `result` and line 4.	All assertions pass.	Pass
TC25	FR2	Detects `+=` string concatenation inside a `while` loop.	One smell detected with target `result` and line 4.	All assertions pass.	Pass

Continued on next page

(Continued from previous page)

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC26	FR2	Detects `+=` modifying a list item inside a loop.	One smell detected with target `self.text[0]` and line 6.	All assertions pass.	Pass
TC27	FR2	Detects `+=` modifying an object attribute inside a loop.	One smell detected with target `self.text` and line 6.	All assertions pass.	Pass
TC28	FR2	Detects `+=` modifying a dictionary value inside a loop.	One smell detected with target `data['key']` and line 4.	All assertions pass.	Pass
TC29	FR2	Detects multiple separate string concatenations in a loop.	Two smells detected with targets `result` and `logs[0]` on line 5.	All assertions pass.	Pass
TC30	FR2	Detects string concatenations with re-assignments inside the loop.	One smell detected with target `result` and line 4.	All assertions pass.	Pass
TC31	FR2	Detects concatenation inside nested loops.	One smell detected with target `result` and line 5.	All assertions pass.	Pass
TC32	FR2	Detects multi-level concatenations belonging to the same smell.	One smell detected with target `result` and two occurrences on lines 4 and 5.	All assertions pass.	Pass
TC33	FR2	Detects `+=` inside an `if-else` condition within a loop.	One smell detected with target `result` and two occurrences on line 4.	All assertions pass.	Pass
TC34	FR2	Detects `+=` using f-strings inside a loop.	One smell detected with target `result` and line 4.	All assertions pass.	Pass

Continued on next page

(Continued from previous page)

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC35	FR2	Detects `+=` using `%` formatting inside a loop.	One smell detected with target `result` and line 4.	All assertions pass.	Pass
TC36	FR2	Detects `+=` using `.format()` inside a loop.	One smell detected with target `result` and line 4.	All assertions pass.	Pass
TC37	FR2	Ensures accessing the concatenation variable inside the loop is NOT flagged.	No smells detected.	All assertions pass.	Pass
TC38	FR2	Ensures regular string assignments are NOT flagged.	No smells detected.	All assertions pass.	Pass
TC39	FR2	Ensures number operations with `+=` are NOT flagged.	No smells detected.	All assertions pass.	Pass
TC40	FR2	Ensures string concatenation OUTSIDE a loop is NOT flagged.	No smells detected.	All assertions pass.	Pass
TC41	FR2	Detects a variable concatenated multiple times in the same loop iteration.	One smell detected with target `result` and two occurrences on line 4.	All assertions pass.	Pass
TC42	FR2	Detects concatenation where both prefix and suffix are added.	One smell detected with target `result` and line 4.	All assertions pass.	Pass
TC43	FR2	Detects `+=` where new values are inserted at the beginning instead of the end.	One smell detected with target `result` and line 4.	All assertions pass.	Pass

Continued on next page

(Continued from previous page)

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC44	FR2	Ignores potential smells where type cannot be confirmed as a string.	No smells detected.	All assertions pass.	Pass
TC45	FR2	Detects string concatenation where type is inferred from function type hints.	One smell detected with target `result` and line 4.	All assertions pass.	Pass
TC46	FR2	Detects string concatenation where type is inferred from variable type hints.	One smell detected with target `result` and line 4.	All assertions pass.	Pass
TC47	FR2	Detects string concatenation where type is inferred from class attributes.	One smell detected with target `result` and line 9.	All assertions pass.	Pass
TC48	FR2	Detects string concatenation where type is inferred from the initial value assigned.	One smell detected with target `result` and line 4.	All assertions pass.	Pass

Table 6: String Concatenation in Loop Detection Test Cases

4.4.2 Long Element Chain Detector Module

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC49	FR2	Test with code that has no chains.	No chains should be detected.	All assertions pass.	Pass
TC50	FR2	Test with chains shorter than threshold.	No chains should be detected for threshold of 5.	All assertions pass.	Pass
TC51	FR2	Test with chains exactly at threshold.	One chain should be detected at line 3.	All assertions pass.	Pass

Continued on next page

(Continued from previous page)

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC52	FR2	Test with chains longer than threshold.	One chain should be detected with message "Dictionary chain too long (4/3)".	All assertions pass.	Pass
TC53	FR2	Test with multiple chains in the same file.	Two chains should be detected at different lines.	All assertions pass.	Pass
TC54	FR2	Test chains inside nested functions and classes.	Two chains should be detected, one inside a function, one inside a class.	All assertions pass.	Pass
TC55	FR2	Test that chains on the same line are reported only once.	One chain should be detected at line 4.	All assertions pass.	Pass
TC56	FR2	Test chains with different variable types.	Two chains should be detected, one in a list and one in a tuple.	All assertions pass.	Pass
TC57	FR2	Test with a custom threshold value.	No chains detected with threshold 4. One chain detected with threshold 2.	All assertions pass.	Pass
TC58	FR2	Test the structure of the returned LECSmell object.	Object should have correct type, path, module, symbol, and occurrence details.	All assertions pass.	Pass
TC59	FR2	Test chains within complex expressions.	Three chains should be detected in different contexts.	All assertions pass.	Pass
TC60	FR2	Test with an empty file.	No chains should be detected.	All assertions pass.	Pass

TC61	FR2	Test with threshold of 1 (every subscript reported).	One chain should be detected with message “Dictionary chain too long (5/5)”.	All assertions pass.	Pass

Table 7: Long Element Chain Detector Module Test Cases

4.4.3 Repeated Calls Detection Module

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC62	FR2, PR-PAR2	Test detection of repeated function calls within the same scope.	One repeated call detected with two occurrences.	All assertions pass.	Pass
TC63	FR2, PR-PAR2	Test detection of repeated method calls on the same object instance.	One repeated method call detected with two occurrences.	All assertions pass.	Pass
TC64	FR2	Test that function calls with different arguments are not flagged.	No repeated calls should be detected.	All assertions pass.	Pass
TC65	FR2	Test that function calls on modified objects are not flagged.	No repeated calls should be detected due to object state change.	All assertions pass.	Pass
TC66	FR2, PR-PAR3	Test detection of repeated external function calls.	One repeated function call detected with two occurrences.	All assertions pass.	Pass
TC67	FR2, PR-PAR3	Test detection of repeated calls to expensive built-in functions.	One repeated function call detected with two occurrences.	All assertions pass.	Pass
TC68	FR2, PR-PAR3	Test that built-in functions with primitive arguments are not flagged.	No repeated calls should be detected.	All assertions pass.	Pass

Continued on next page

(Continued from previous page)

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC69	FR2	Test that method calls on different object instances are not flagged.	No repeated calls should be detected.	All assertions pass.	Pass

Table 8: Repeated Calls Detection Module Test Cases

4.4.4 Long Lambda Element Detection Module

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC70	FR2	Test code with no lambdas.	No smells should be detected.	All assertions pass.	Pass
TC71	FR2	Test short single lambda (under thresholds).	No smells should be detected.	All assertions pass.	Pass
TC72	FR2	Test lambda exceeding expression count threshold.	One smell should be detected.	All assertions pass.	Pass
TC73	FR2	Test lambda exceeding character length threshold (100).	One smell should be detected.	All assertions pass.	Pass
TC74	FR2	Test lambda exceeding both expression and length thresholds.	At least one smell should be detected.	All assertions pass.	Pass
TC75	FR2	Test nested lambdas.	Two smells should be detected.	All assertions pass.	Pass
TC76	FR2	Test inline lambdas passed to functions.	Two smells should be detected.	All assertions pass.	Pass
TC77	FR2	Test trivial lambda with no body.	No smells should be detected.	All assertions pass.	Pass

Table 9: Long Lambda Element Detector Module Test Cases

4.4.5 Long Message Chain Detector Module

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC78	FR2	Test chain with exactly five method calls.	One smell should be detected.	All assertions pass.	Pass
TC79	FR2	Test chain with six method calls.	One smell should be detected.	All assertions pass.	Pass
TC80	FR2	Test chain with four method calls.	No smells should be detected.	All assertions pass.	Pass
TC81	FR2	Test chain with both attribute and method calls.	One smell should be detected.	All assertions pass.	Pass
TC82	FR2	Test chain inside a loop.	One smell should be detected.	All assertions pass.	Pass
TC83	FR2	Test multiple chains on the same line.	One smell should be detected.	All assertions pass.	Pass
TC84	FR2	Test separate statements with fewer calls.	No smells should be detected.	All assertions pass.	Pass
TC85	FR2	Test short chain in a comprehension.	No smells should be detected.	All assertions pass.	Pass
TC86	FR2	Test long chain in a comprehension.	One smell should be detected.	All assertions pass.	Pass
TC87	FR2	Test five separate long chains in one function.	Five smells should be detected.	All assertions pass.	Pass
TC88	FR2	Test chain with attribute and index lookups (no calls).	No smells should be detected.	All assertions pass.	Pass
TC89	FR2	Test chain with slicing.	One smell should be detected.	All assertions pass.	Pass
TC90	FR2	Test multiline chain.	One smell should be detected.	All assertions pass.	Pass
TC91	FR2	Test chain inside a lambda.	One smell should be detected.	All assertions pass.	Pass

Continued on next page

(Continued from previous page)

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC92	FR2	Test chain with mixed return types.	One smell should be detected.	All assertions pass.	Pass
TC93	FR2	Test multiple short chains on the same line.	No smells should be detected.	All assertions pass.	Pass
TC94	FR2	Test chain inside a conditional (ternary).	No smells should be detected.	All assertions pass.	Pass

Table 10: Long Message Chain Detector Module Test Cases

4.5 Refactorer Controller Module

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC95	FR5	User requests to refactor a smell.	Correct smell is identified. Logger logs "Running refactoring for long-element-chain using TestRefactorer." Correct refactorer is called once with correct arguments. Output path is `test_path.LEC001_1.py`.	All assertions pass.	Pass
TC96	UHR-UPLD1	System handles missing refactorer.	Raises `NotImplementedError` with message "No refactorer implemented for smell: long-element-chain." Logger logs error.	All assertions pass.	Pass

Continued on next page

(Continued from previous page)

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC97	FR5	Multiple refactorer calls are handled correctly.	Correct smell counter incremented. Refactorer is called twice. First output: `test_path.LEC001_1.py`. Second output: `test_path.LEC001_2.py`.	All assertions pass.	Pass
TC98	FR5	Refactorer runs with overwrite set to False.	Refactorer is called once. Overwrite argument is set to False.	All assertions pass.	Pass
TC99	PR-RFT 1, FR5	System handles empty modified files correctly.	Modified files list remains empty (`[]` in output).	All assertions pass.	Pass

Table 11: Refactorer Controller Module Test Cases

4.6 Smell Refactorers

4.6.1 String Concatenation in Loop

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC100	FR3, FR6	Refactors empty initial concatenation variable (e.g., `result = ""`).	Code is refactored to use a list and `join()`.	All assertions pass.	Pass
TC101	FR3, FR6	Refactors non-empty initial concatenation variable not referenced before the loop.	Code is refactored to use a list and `join()`.	All assertions pass.	Pass
TC102	FR3, FR6	Refactors non-empty initial concatenation variable referenced before the loop.	Code is refactored to use a list and `join()`.	All assertions pass.	Pass

Continued on next page

(Continued from previous page)

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC103	FR3, FR6	Refactors concatenation where the target is not a simple variable (e.g., `result["key"]`).	Code is refactored to use a temporary list and `join()`.	All assertions pass.	Pass
TC104	FR3, FR6	Refactors concatenation where the variable is not initialized in the same scope.	Code is refactored to use a list and `join()`.	All assertions pass.	Pass
TC105	FR3, FR6	Refactors prefix concatenation (e.g., `result = str(i)+ result`).	Code uses `insert(0, ...)` for prefix concatenation.	All assertions pass.	Pass
TC106	FR3, FR6	Refactors concatenation with both prefix and suffix.	Code uses both `insert(0, ...)` and `append(...)`.	All assertions pass.	Pass
TC107	FR3, FR6	Refactors multiple concatenations in the same loop.	Code uses `append(...)` and `insert(0, ...)` as needed.	All assertions pass.	Pass
TC108	FR3, FR6	Refactors nested concatenation in loops.	Code uses `append(...)` and `insert(0, ...)` for nested loops.	All assertions pass.	Pass
TC109	FR3, FR6	Refactors multiple occurrences of concatenation at different loop levels.	Code uses `append(...)` for all occurrences.	All assertions pass.	Pass
TC110	FR3, FR6	Handles reassignment of the concatenation variable inside the loop.	Code resets the list to the new value.	All assertions pass.	Pass
TC111	FR3, FR6	Handles reassignment of the concatenation variable to an empty value.	Code clears the list using `clear()`.	All assertions pass.	Pass

Continued on next page

(Continued from previous page)

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC112	FR3, FR6	Ensures unrelated code and comments are preserved during refactoring.	Unrelated lines and comments remain unchanged.	All assertions pass.	Pass

Table 12: String Concatenation in Loop Refactoring Test Cases

4.6.2 Member Ignoring Method

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC113	FR3, FR6	Refactors a basic member-ignoring method.	Adds `@staticmethod`, removes `self`, and updates calls.	All assertions pass.	Pass
TC114	FR3, FR6	Refactors a member-ignoring method with inheritance.	Updates calls from subclass instances.	All assertions pass.	Pass
TC115	FR3, FR6	Refactors a member-ignoring method with subclass in a separate file.	Updates calls from subclass instances in external files.	All assertions pass.	Pass
TC116	FR3, FR6	Refactors a member-ignoring method with subclass method override.	Does not update calls to overridden methods.	All assertions pass.	Pass
TC117	FR3, FR6	Refactors a member-ignoring method with type hints.	Updates calls using type hints to infer instance type.	All assertions pass.	Pass

Table 13: Member Ignoring Method Refactoring Test Cases

4.6.3 Long Element Chain Refactorer Module

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC118	PR-PAR3, FR6, FR3	Test the long element chain refactorer on basic nested dictionary access	Dictionary should be flattened, and access updated	Refactoring applied successfully, dictionary access updated	Pass
TC119	PR-PAR3, FR6, FR3	Test the long element chain refactorer across multiple files	Dictionary access across multiple files should be updated	Refactoring applied successfully across multiple files	TBD
TC120	PR-PAR3, FR6, FR3	Test the refactorer on dictionary access via class attributes	Class attributes should be flattened and access updated	Refactoring applied successfully on class attribute accesses. All accesses changed correctly.	Pass
TC121	PR-PAR3, FR6, FR3	Ensure the refactorer skips shallow dictionary access	Refactoring should be skipped for shallow access	Refactoring correctly skipped for shallow access	Pass
TC122	PR-PAR3, FR6, FR3	Test the refactorer on dictionary access with mixed depths	Flatten the dictionary up to the minimum access depth	All dictionary access chains flattened to minimum access depth and dictionary flattened successfully.	Pass

Table 14: Long Element Chain Refactorer Test Cases

4.6.4 Repeated Calls Refactoring Module

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC123	FR3, FR5, PR-PAR3	Test that repeated function calls are cached properly.	The function calls should be replaced with a cached variable.	All assertions pass.	Pass
TC124	FR3, FR5, PR-PAR3	Test that repeated method calls on the same object are cached.	Method calls should be replaced with a cached result stored in a variable.	All assertions pass.	Pass
TC125	FR3, FR5, PR-PAR2	Test that repeated method calls on different object instances are not cached.	Calls on different object instances should remain unchanged.	All assertions pass.	Pass
TC126	FR3, FR5	Test that caching is applied even with multiple identical function calls.	The repeated function calls should be replaced with a cached variable.	All assertions pass.	Pass
TC127	FR3, FR5	Test caching when refactoring function calls that appear in a docstring.	Function calls inside the docstring should not be modified.	All assertions pass.	Pass
TC128	FR3, FR5, PR-PAR3	Test caching of method calls inside a class with an unchanged instance state.	Repeated method calls should be cached correctly.	All assertions pass.	Pass
TC129	FR3, FR5	Test that functions with varying arguments are not cached.	Calls with different arguments should remain unchanged.	All assertions pass.	Pass
TC130	FR3, FR5, PR-PAR2	Test that caching does not interfere with scope and closures.	The cached value should remain valid within the correct scope.	All assertions pass.	Pass

Table 15: Cache Repeated Calls Refactoring Module Test Cases

4.6.5 Use a Generator Refactoring Module

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC131	FR3, FR5, PR-PAR3	Test refactoring of list comprehensions in ‘all()‘ calls.	The list comprehension should be converted into a generator expression.	All assertions pass.	Pass
TC132	FR3, FR5, PR-PAR3	Test refactoring of list comprehensions in ‘any()‘ calls.	The list comprehension should be converted into a generator expression.	All assertions pass.	Pass
TC133	FR3, FR5, PR-PAR3	Test refactoring of multi-line list comprehensions.	The multi-line comprehension should be refactored correctly while preserving indentation.	All assertions pass.	Pass
TC134	FR3, FR5, PR-PAR3	Test refactoring of complex conditions within ‘any()‘ and ‘all()‘.	The refactored generator expression should maintain logical correctness.	All assertions pass.	Pass
TC135	FR3, FR5	Test that improperly formatted list comprehensions are handled correctly.	No unintended modifications should be applied to non-standard formats.	All assertions pass.	Pass
TC136	FR3, FR5	Test that readability is preserved in refactored code.	The refactored code should be clear, well-formatted, and maintain original intent.	All assertions pass.	Pass
TC137	FR3, FR5	Test that list comprehensions outside of ‘all()‘ and ‘any()‘ remain unchanged.	The refactorer should not modify list comprehensions used in other contexts.	All assertions pass.	Pass
TC138	FR3, FR5	Test refactoring when ‘all()‘ or ‘any()‘ calls are nested.	The refactored code should handle nested expressions correctly.	All assertions pass.	Pass

Table 16: Use a Generator Refactoring Module Test Cases

4.6.6 Long Lambda Element Refactorer

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC139	FR1, FR2, FR3, FR5, FR6	Refactor a basic single-line lambda.	Lambda is converted to a named function.	All assertions pass.	Pass
TC140	FR1, FR2, FR3, FR5, FR6	Ensure no print statements are added unnecessarily.	Refactored code contains no print statements.	All assertions pass.	Pass
TC141	FR1, FR2, FR3, FR5, FR6	Refactor a lambda passed as an argument to another function.	Lambda is converted to a named function and used correctly.	All assertions pass.	Pass
TC142	FR1, FR2, FR3, FR5, FR6	Refactor a lambda with multiple parameters.	Lambda is converted to a named function with multiple parameters.	All assertions pass.	Pass
TC143	FR1, FR2, FR3, FR5, FR6	Refactor a lambda used with keyword arguments.	Lambda is converted to a named function and used correctly with keyword arguments.	All assertions pass.	Pass
TC144	FR1, FR2, FR3, FR5, FR6	Refactor a very long lambda spanning multiple lines.	Lambda is converted to a named function preserving the logic.	All assertions pass.	Pass

Table 17: Long Lambda Element Refactorer Test Cases

4.6.7 Long Message Chain Refactorer

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC145	FR1, FR2, FR3, FR5, FR6	Refactor a basic method chain.	Method chain is split into intermediate variables.	All assertions pass.	Pass
TC146	FR1, FR2, FR3, FR5, FR6	Refactor a long message chain with an f-string.	F-string chain is split into intermediate variables.	All assertions pass.	Pass
TC147	FR1, FR2, FR3, FR5, FR6	Ensure modifications occur even if the method chain isn't long.	Short method chain is split into intermediate variables.	All assertions pass.	Pass
TC148	FR1, FR2, FR3, FR5, FR6	Ensure indentation is preserved after refactoring.	Refactored code maintains proper indentation.	All assertions pass.	Pass
TC149	FR1, FR2, FR3, FR5, FR6	Refactor method chains containing method arguments.	Method chain with arguments is split into intermediate variables.	All assertions pass.	Pass
TC150	FR1, FR2, FR3, FR5, FR6	Refactor print statements with method chains.	Print statement with method chain is split into intermediate variables.	All assertions pass.	Pass
TC151	FR1, FR2, FR3, FR5, FR6	Refactor nested method chains.	Nested method chain is split into intermediate variables.	All assertions pass.	Pass

Table 18: Long Message Chain Refactorer Test Cases

4.6.8 Long Parameter List

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC152	FR3, FR6	Refactors a constructor definition with 8 parameters, and class initialization with positional arguments.	Declares grouping classes. Updates constructor call with grouped instantiations. Also updates function signature and body to reflect new parameters.	All assertions pass.	Pass
TC153	FR3, FR6	Refactors a constructor definition with 8 parameters with one unused in body, as well as class initialization with positional arguments.	Declares grouping classes. Updates constructor call with grouped instantiations. Also updates function signature and body to reflect new used parameters.	All assertions pass.	Pass
TC154	FR3, FR6	Refactors an instance method with 8 parameters (two default values) and the call made to it (1 positional argument).	Declares grouping classes with default values preserved. Updates method call with grouped instantiations. Also updates method signature and body to reflect new parameters.	All assertions pass.	Pass

Continued on next page

(Continued from previous page)

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC155	FR3, FR6	Refactors a static method with 8 parameters (1 with default value, 4 unused in body) and the call made to it (2 positional arguments)	Declares grouping classes with default values preserved. Updates method call with grouped instantiations. Also updates method signature and body to reflect new used parameters.	All assertions pass.	Pass
TC156	FR3, FR6	Refactors a standalone function with 8 parameters (1 with default value that is also unused in body) and the call made to it (1 positional arguments)	Declares grouping classes. Updates method call with grouped instantiations. Also updates method signature and body to reflect new used parameters.	All assertions pass.	Pass

Table 19: Long Parameter List Refactoring Test Cases

4.7 VS Code Extension

4.7.1 Detect Smells Command

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC157	FR10, OER-IAS1	No active editor is found.	Shows error message: "Eco: No active editor found."	All assertions pass.	Pass
TC158	FR10, OER-IAS1	Active editor has no valid file path.	Shows error message: "Eco: Active editor has no valid file path."	All assertions pass.	Pass

Continued on next page

(Continued from previous page)

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC159	FR10, OER-IAS1	No smells are enabled.	Shows warning message: "Eco: No smells are enabled! Detection skipped."	All assertions pass.	Pass
TC160	FR10, OER-IAS1	Uses cached smells when hash is unchanged and same smells are enabled.	Shows info message: "Eco: Using cached smells for fake.path"	All assertions pass.	Pass
TC161	FR10, OER-IAS1	Fetches new smells when enabled smells change.	Calls `wipeWorkCache`, `updateHash`, and `fetchSmells`. Updates workspace data.	All assertions pass.	Pass
TC162	FR10, OER-IAS1	Fetches new smells when hash changes but enabled smells remain the same.	Calls `updateHash` and `fetchSmells`. Updates workspace data.	All assertions pass.	Pass
TC163	FR10, OER-IAS1	No cached smells and server is down.	Shows warning message: "Action blocked: Server is down and no cached smells exist for this file version."	All assertions pass.	Pass
TC164	FR10, OER-IAS1	Highlights smells when smells are found.	Shows info messages and calls `highlightSmells`.	All assertions pass.	Pass

Table 20: Detect Smells Command Test Cases

4.7.2 Refactor Smell Command

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC165	PR-RFT1	No active editor is found.	Shows error message "Eco: Unable to proceed as no active editor or file path found."	All assertions pass.	Pass
TC166	PR-RFT1, FR6	Attempting to refactor when no smells are detected in the file	Shows error message "Eco: No smells detected in the file for refactoring."	All assertions pass.	Pass
TC167	FR6	Attempting to refactor when selected line doesn't match any smell	Shows error message "Eco: No matching smell found for refactoring."	All assertions pass.	Pass
TC168	FR5, FR6, FR10	Refactoring a smell when found on the selected line	Saves the current file. Calls `refactorSmell` method with correct parameters. Shows message "Refactoring report available in sidebar". Executes command to focus refactor sidebar. Opens and shows the refactored preview. Highlights updated smells. Updates the UI with new smells	All assertions pass.	Pass
TC169	PR-RFT2	Handling API failure during refactoring	Shows error message "Eco: Refactoring failed. See console for details."	All assertions pass.	Pass

Table 21: Refactor Smell Command Test Cases

4.7.3 File Highlighter

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC170	FR10, OER-IAS1, LFR-AP2	Creates decorations for a given color.	Decoration is created using `vscode.window.createTextEditorDecorationType`.	All assertions pass.	Pass
TC171	FR10, OER-IAS1, LFR-AP2	Highlights smells in the active text editor.	Decorations are set using `setDecorations`.	All assertions pass.	Pass
TC172	FR10, OER-IAS1, LFR-AP2	Does not reset highlight decorations on first initialization.	Decorations are not disposed of on the first call.	All assertions pass.	Pass
TC173	FR10, OER-IAS1, LFR-AP2	Resets highlight decorations on subsequent calls.	Decorations are disposed of on subsequent calls.	All assertions pass.	Pass

Table 22: File Highlighter Test Cases

4.7.4 File Hashing

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC174	FR10, OER-IAS1	Document hash has not changed.	Does not update workspace storage.	All assertions pass.	Pass
TC175	FR10, OER-IAS1	Document hash has changed.	Updates workspace storage.	All assertions pass.	Pass

TC176	FR10, OER-IAS1	No hash exists for the document.	Updates workspace storage.	All assertions pass.	Pass

Table 23: Hashing Tools Test Cases

4.7.5 Line Selection Manager Module

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC177	UHR-EOU1	Call the ‘removeLastComment‘ method after adding a comment.	The decoration is removed and no comment remains on the line.	The decoration is removed, and no comment appears on the selected line.	Pass
TC178	UHR-EOU1	Call ‘commentLine‘ method with null editor.	The method does not throw an error.	The method does not throw an error.	Pass
TC179	UHR-EOU1	Call ‘commentLine‘ on a file with no detected smells.	No comment is added to the line.	No decoration is added, and the line remains unchanged.	Pass
TC180	UHR-EOU1	Call ‘commentLine‘ on a file where the document hash does not match.	The method does not add a comment because the document has changed.	No decoration is added due to the document hash mismatch.	Pass
TC181	UHR-EOU1	Call ‘commentLine‘ with a multi-line selection.	The method returns early without adding a comment.	No comment is added to any lines in the selection.	Pass
TC182	UHR-EOU1	Call ‘commentLine‘ on a line with no detected smells.	No comment is added for the line.	No decoration is added, and the line remains unchanged.	Pass
TC183	UHR-EOU1	Call ‘commentLine‘ on a line with a single detected smell.	The comment shows the first smell symbol without a count.	Comment shows the first smell symbol: ‘Smell: PERF-001‘.	Pass

Continued on next page

(Continued from previous page)

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC184	UHR-EOU1	Call ‘commentLine‘ on a line with a detected smell.	A comment is added on the selected line in the editor showing the detected smell.	Comment added with the correct smell symbol and count.	Pass
TC185	UHR-EOU1	Call ‘commentLine‘ on a line with multiple detected smells.	The comment shows the first smell followed by the count of additional smells.	Comment shows ‘Smell: PERF-001 — (+1)‘.	Pass

Table 24: Line Selection Module Test Cases

4.7.6 Hover Manager Module

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC186	LFR-AP2	Register hover provider for Python files.	Hover provider registered for Python files.	Hover provider is registered for Python files.	Pass
TC187	LFR-AP2	Subscribe hover provider.	Hover provider subscription registered.	Hover provider subscription registered.	Pass
TC188	LFR-AP2	Return hover content with no smells.	Returns null for hover content.	Hover content = null.	Pass
TC189	LFR-AP2, FR2	Update smells with new data.	Smells updated correctly with new data.	Smells are updated correctly with new smells data.	Pass
TC190	LFR-AP2, FR2	Update smells correctly.	Smells updated with new content.	Current smells content updated to new smells content provided.	Pass

Continued on next page

(Continued from previous page)

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC191	LFR-AP2	Generate valid hover content.	Generates hover content with correct smell information.	Correct and valid hover content generated for given smell.	Pass
TC192	LFR-AP2	Register refactor commands.	Both commands registered correctly on initialization	Refactor commands registered correctly.	Pass

Table 25: Hover Manager Module Test Cases

4.7.7 Handle Smell Settings Module

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC193	FR10, UHR-PSI1	Test retrieval of enabled smells from settings.	Function should return the current enabled smells.	All assertions pass.	Pass
TC194	FR10, UHR-PSI1	Test retrieval of enabled smells when no settings exist.	Function should return an empty object.	All assertions pass.	Pass
TC195	FR10, UHR-PSI1, UHR-EOU2	Test enabling a smell and verifying notification.	Notification should be displayed and cache wiped.	All assertions pass.	Pass
TC196	FR10, UHR-PSI1, UHR-EOU2	Test disabling a smell and verifying notification.	Notification should be displayed and cache wiped.	All assertions pass.	Pass

Continued on next page

(Continued from previous page)

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC197	FR10, UHR-PSI1, UHR-EOU2	Test that cache is not wiped if no changes occur.	No notification or cache wipe should happen.	All assertions pass.	Pass
TC198	FR10, UHR-PSI1	Test formatting of kebab-case smell names.	Smell names should be correctly converted to readable format.	All assertions pass.	Pass
TC199	FR10, UHR-PSI1	Test formatting with an empty string input.	Function should return an empty string without errors.	All assertions pass.	Pass

Table 26: VS Code Settings Management Module Test Cases

4.7.8 Handle Smell Settings Module

4.7.9 Wipe Workspace Cache Command

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC200	FR5, FR8	Trigger the cache wipe with no reason provided.	The smells cache should be cleared and reset to an empty state. A success message indicating that the workspace cache was successfully wiped should be displayed.	All assertions pass.	Pass

Continued on next page

(Continued from previous page)

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC201	FR5, FR8	Trigger the cache wipe with the reason "manual".	Both the smells cache and file changes cache should be cleared and reset to empty states. A success message indicating that the workspace cache was manually wiped by the user should be displayed.	All assertions pass.	Pass
TC202	FR5, FR8	Trigger the cache wipe when there are no open files.	A log message indicating that there are no open files to update should be generated.	All assertions pass.	Pass
TC203	FR5, FR8	Trigger the cache wipe when there are open files.	A message indicating the number of visible files should be logged, and the hashes for each open file should be updated.	All assertions pass.	Pass
TC204	FR5, FR8	Trigger the cache wipe with the reason "settings".	Only the smells cache should be cleared. A success message indicating that the cache was wiped due to smell detection settings changes should be displayed.	All assertions pass.	Pass
TC205	FR3, FR5, FR8	Trigger the cache wipe when an error occurs.	An error message should be logged, and an error message indicating failure to wipe the workspace cache should be displayed to the user.	All assertions pass.	Pass

Table 27: Wipe Workspace Cache Command Test Cases

4.7.10 Backend

ID	Ref. Req.	Action	Expected Result	Actual Result	Result
TC206	PR-SCR1, PR-RFT1	Trigger request to check server status when server responds with a successful status.	The set status method should be called with the `ServerStatusType.UP` status	All assertions pass.	Pass
TC207	PR-SCR1, PR-RFT2	Trigger request to check server status when server responds with an error status or fails to respond.	The set status method should be called with the `ServerStatusType.DOWN` status	All assertions pass.	Pass
TC208	FR-6, PR-RFT1	Trigger intialize logs call with a valid directory path, and backend responds with success.	The function should return `true` indicating successful log initialization.	All assertions pass.	Pass
TC209	PR-SCR1, PR-RFT2	Trigger intialize logs call with a valid directory path, and backend responds with a failure.	TThe function should return `false` indicating failure to initialize logs.	All assertions pass.	Pass

Table 28: Backend Test Cases

5 Changes Due to Testing

During the testing phase, several changes were made to the tool based on feedback from user testing, supervisor reviews, and edge cases encountered during unit and integration testing. These changes were necessary to improve the tool's usability, functionality, and robustness.

5.1 Usability and User Input Adjustments

One of the key findings from testing was the balance between **automating refactorings** and **allowing user control** over changes. Initially, the tool required users to manually approve every refactoring, which slowed down the workflow. However, after usability testing, it became evident that an **option to refactor all occurrences of the same smell type** would significantly improve efficiency. This led to the introduction of a **"Refactor Smell of Same Type"** feature in the VS Code extension, allowing users to apply the same refactoring across multiple instances of a detected smell simultaneously. Additionally, we refined the

Accept/Reject UI elements to make them more intuitive and streamlined the workflow for batch refactoring actions.

5.2 Detection and Refactoring Improvements

Heavy modifications were made to the **detection and refactoring modules**, particularly in handling **multi-file projects**. Initially, the detectors and refactorers assumed a **single-file scope**, leading to missed optimizations when function calls or variable dependencies spanned across multiple files. After extensive testing, the detection system was updated to track **cross-file dependencies**, ensuring that refactoring suggestions accounted for the broader codebase.

5.3 VS Code Extension Enhancements

Through usability testing, it became apparent that **integrating the tool as a VS Code extension** was a significant improvement over a standalone CLI tool. This led to the following enhancements:

- **Enhanced Hover Tooltips** – Descriptions for detected code smells were rewritten to be clearer and more informative.
- **Smell Filtering Options** – Users can now enable or disable specific code smell detections directly from the VS Code settings menu.

5.4 Future Revisions and Remaining Work

Certain features, including **report generation and full documentation availability**, have yet to be fully implemented. These components will be finalized in **Revision 1**, where testing will ensure that:

- The **reporting system correctly logs detected smells, applied refactorings, and energy savings**.
- The **documentation includes detailed installation, usage, and troubleshooting guides**.

Additionally, once all features are complete, the **VS Code extension will be packaged and tested as a full release** to ensure seamless installation via the VS Code Marketplace.

Overall, the testing phase played a crucial role in refining the tool's functionality, optimizing performance, and improving usability. The feedback gathered led to meaningful changes that enhance both the developer experience and the effectiveness of automated refactoring.

6 Automated Testing

All test for the Python backend as well as the individual modules on the TypeScript side (for the VSCode extension) are automated. The Python tests are run using Pytest by simply typing `pytest` in the command line in the root project directory. All the Typescript tests can be run similarly, though they run with Jest and through the command `npm run test`. The results for both are printed to the console.

7 Trace to Requirements

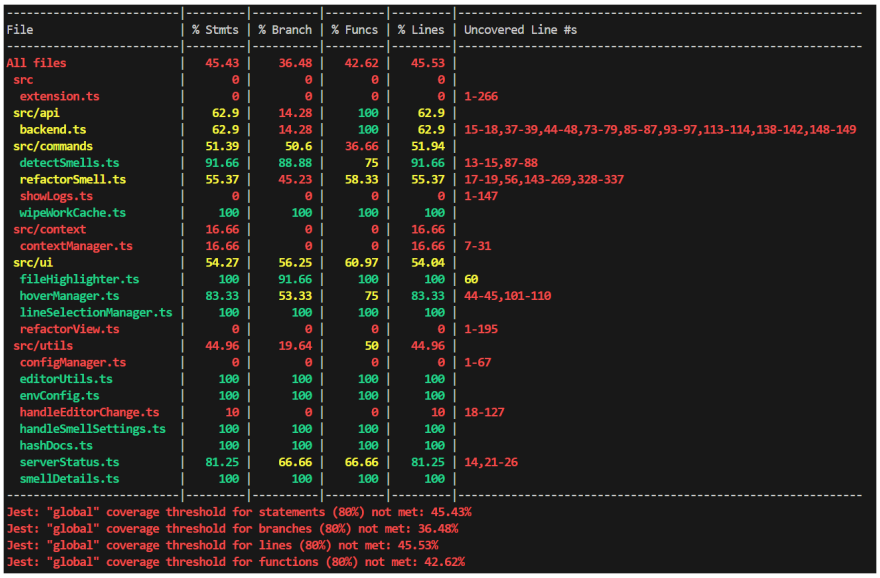
8 Trace to Modules

9 Code Coverage Metrics

The following analyzes the code coverage metrics for the Python backend and frontend (TypeScript) of the VSCode extension. The analysis is based on the coverage data provided in Figure 11 (Python backend) and Figure 12 (frontend). Code coverage is a measure of how well the codebase is tested, and it helps identify areas that may require additional testing.

```
---------- coverage: platform win32, python 3.10.11-final-0 ----------
Name                                                                                    Stmts   Miss  Cover
-----------------------------------------------------------------------------------------------------------
src\ecooptimizer\analyzers\analyzer_controller.py                                          82      4    95%
src\ecooptimizer\analyzers\ast_analyzers\detect_long_element_chain.py                      26      0   100%
src\ecooptimizer\analyzers\ast_analyzers\detect_long_lambda_expression.py                  37      1    97%
src\ecooptimizer\analyzers\ast_analyzers\detect_long_message_chain.py                      27      1    96%
src\ecooptimizer\analyzers\ast_analyzers\detect_repeated_calls.py                          70      2    97%
src\ecooptimizer\analyzers\astroid_analyzers\detect_string_concat_in_loop.py              140      6    96%
src\ecooptimizer\api\routes\detect_smells.py                                               38      0   100%
src\ecooptimizer\api\routes\refactor_smell.py                                             103      8    92%
src\ecooptimizer\api\routes\show_logs.py                                                   62     42    32%
src\ecooptimizer\config.py                                                                 10      0   100%
src\ecooptimizer\data_types\custom_fields.py                                               18      0   100%
src\ecooptimizer\data_types\smell.py                                                       25      0   100%
src\ecooptimizer\data_types\smell_record.py                                                 9      0   100%
src\ecooptimizer\exceptions.py                                                             11      2    82%
src\ecooptimizer\measurements\base_energy_meter.py                                          8      1    88%
src\ecooptimizer\measurements\codecarbon_energy_meter.py                                   42      2    95%
src\ecooptimizer\refactorers\base_refactorer.py                                            11      1    91%
src\ecooptimizer\refactorers\concrete\list_comp_any_all.py                                 40      2    95%
src\ecooptimizer\refactorers\concrete\long_element_chain.py                               185      6    97%
src\ecooptimizer\refactorers\concrete\long_lambda_function.py                              69      7    90%
src\ecooptimizer\refactorers\concrete\long_message_chain.py                                57      4    93%
src\ecooptimizer\refactorers\concrete\long_parameter_list.py                              284     41    86%
src\ecooptimizer\refactorers\concrete\member_ignoring_method.py                           146      5    97%
src\ecooptimizer\refactorers\concrete\repeated_calls.py                                    87      9    90%
src\ecooptimizer\refactorers\concrete\str_concat_in_loop.py                               176     19    89%
src\ecooptimizer\refactorers\multi_file_refactorer.py                                      49      9    82%
src\ecooptimizer\refactorers\refactorer_controller.py                                      24      0   100%
tests\conftest.py                                                                           8      1    88%
-----------------------------------------------------------------------------------------------------------
TOTAL                                                                                    1844    173    91%
```

Figure 11: Coverage Report of the Python Backend Library



File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	45.43	36.48	42.62	45.53	
src	0	0	0	0	
extension.ts	0	0	0	0	1-266
src/api	62.9	14.28	100	62.9	
backend.ts	62.9	14.28	100	62.9	15-18,37-39,44-48,73-79,85-87,93-97,113-114,138-142,148-149
src/commands	51.39	50.6	36.66	51.94	
detectSmells.ts	91.66	88.88	75	91.66	13-15,87-88
refactorSmell.ts	55.37	45.23	58.33	55.37	17-19,56,143-269,328-337
showLogs.ts	0	0	0	0	1-147
wipeWorkCache.ts	100	100	100	100	
src/context	16.66	0	0	16.66	
contextManager.ts	16.66	0	0	16.66	7-31
src/ui	54.27	56.25	60.97	54.04	
fileHighlighter.ts	100	91.66	100	100	60
hoverManager.ts	83.33	53.33	75	83.33	44-45,101-110
lineSelectionManager.ts	100	100	100	100	
refactorView.ts	0	0	0	0	1-195
src/utils	44.96	19.64	50	44.96	
configManager.ts	0	0	0	0	1-67
editorUtils.ts	100	100	100	100	
envConfig.ts	100	100	100	100	
handleEditorChange.ts	10	0	0	10	18-127
handleSmellSettings.ts	100	100	100	100	
hashDocs.ts	100	100	100	100	
serverStatus.ts	81.25	66.66	66.66	81.25	14,21-26
smellDetails.ts	100	100	100	100	

Jest: "global" coverage threshold for statements (80%) not met: 45.43%
Jest: "global" coverage threshold for branches (80%) not met: 36.48%
Jest: "global" coverage threshold for lines (80%) not met: 45.53%
Jest: "global" coverage threshold for functions (80%) not met: 42.62%

Figure 12: Coverage Report of the VSCode Extension

9.1 VSCode Extension

The frontend codebase has an overall coverage of 45.43% for statements, 36.48% for branches, 42.62% for functions, and 45.53% for lines (Figure 12). These metrics fall below the global coverage thresholds of 80% for the following reasons. The file `extension.ts`, which contains the core logic for the VSCode extension, has 0% coverage as it is mainly made up of initialization commands with no real logic that can be tested. The file `refactorView.ts`, responsible for the refactoring view, also has 0% coverage. This module is a UI component and will be tested for revision 1. Since `handleEditorChange.ts` is closely related to the UI component, its testing has also been put off.

The file `refactorSmell.ts` has moderate coverage (55.37% statements, 45.23% branches), with significant gaps in testing around lines 143–269 and 328–337 (Figure 12). This is due to a feature that is not fully implemented and therefore not tested. Finally, `configManager.ts` has not been tested as yet due to evolving configuration options, but will be tested for revision 1.

9.2 Python Backend

The backend codebase has an overall coverage of 91% (Figure 11) and has been thoroughly tested as it contains the key features of project and the bulk of the logic. The exception is `show_logs.py`, which handles the websocket endpoint for logging, due to the complex nature of this module testing has been omitted. Since its function is mainly to broadcast logs it is also relatively simple to verify its functionality manually

Appendix A – Usability Testing Data

Protocol

Purpose

The purpose of this usability test is to evaluate the ease of use, efficiency, and overall user experience of the VSCode extension for refactoring Python code to improve energy efficiency. The test will identify usability issues that may hinder adoption by software developers.

Objective

Evaluate the usability of the extension's **smell detection**, **refactoring process**, **customization settings**, and **refactoring view**.

- Assess how easily developers can navigate the extension interface.
- Measure the efficiency of the workflow when applying or rejecting refactorings.
- Identify areas of confusion or frustration.

Methodology

Test Type

Moderated usability testing.

Participants

- **Target Users:** Python developers who use VSCode.
- **Number of Participants:** 5–7.
- **Recruitment Criteria:**
  - Experience with Python development.
  - Familiarity with VSCode.
  - No prior experience with this extension.

Testing Environment

- **Hardware:** Provided computer.
- **Software:**
  - VSCode (latest stable release).

  - The VSCode extension installed.
  - Screen recording software (optional, for post-test analysis).
  - A sample project with **predefined code snippets** containing various **code smells**.
- **Network Requirements:** Stable internet connection for remote testing.

Test Moderator Role

- Introduce the test and explain objectives.
- Observe user interactions without providing assistance unless necessary.
- Take notes on usability issues, pain points, and confusion.
- Ask follow-up questions after each task.
- Encourage participants to **think aloud**.

Data Collection

Metrics

- **Task Success Rate:** Percentage of users who complete tasks without assistance.
- **Error Rate:** Number of errors or missteps per task.
- **User Satisfaction:** Post-test rating on a scale of 1–5.

Qualitative Data

- Observations of confusion, hesitation, or frustration.
- Participant comments and feedback.
- Follow-up questions about expectations vs. actual experience.
- Pre-test survey.
- Post-test survey.

Analysis and Reporting

- Identify common pain points and recurring issues.
- Categorize usability issues by severity:
  - **Critical:** Blocks users from completing tasks.
  - **Major:** Causes significant frustration but has workarounds.
  - **Minor:** Slight inconvenience, but doesn't impact core functionality.
- Provide recommendations for UI/UX improvements.
- Summarize key findings and next steps.

Next Steps

- Fix major usability issues before release.
- Conduct follow-up usability tests if significant changes are made.
- Gather further feedback from real users post-release.

Task List

Mock Installation Documentation

The extension can be installed to detect energy inefficiencies (smells) in your code and refactor them.

Commands

Open the VSCode command palette (`CTRL+SHIFT+P`):

- **Detect Smells:** `Eco: Detect Smells`
- **Refactor Smells:** `Eco: Refactor Smell` or `CTRL+SHIFT+R` (or to be discovered).

Tasks

Report your observations **aloud**!

Task 1: Smell Detection

1. Open the `sample.py` file.
2. Detect the smells in the file.
3. What do you see?

Task 2: Line Selection

1. In the same `sample.py` file, select one of the highlighted lines.
2. What do you see?
3. Select another line.

Task 3: Hover

1. In the same file, hover over a highlighted line.
2. What do you see?

Task 4: Initiate Refactoring (Single)

1. In the same file, refactor any smell of your choice.
2. What do you observe immediately after?
3. Does a sidebar pop up after some time?

Task 5: Refactor Smell (Sidebar)

1. What information do you see in the sidebar?
2. Do you understand the information communicated?
3. Do you see what was changed in the file?
4. Try rejecting a smell. Did the file change?
5. Repeat Tasks 1, 4, and 5, but reject a smell. Did the file stay the same?

Task 6: Refactor Multi-File Smell

1. Open the `main.py` file.
2. Detect the smells in the file.
3. Refactor any smell of your choice.
4. Do you see anything different in the sidebar?
5. Try clicking on the new addition to the sidebar. Notice anything?
6. Try accepting the refactoring. Did both files change?

Task 7: Change Smell Settings

1. Open the `sample.py` file.
2. Detect the smells in the file.
3. Take note of the smells detected.
4. Open the settings page (`CTRL+,`).
5. Navigate to the **Extensions** drop-down and select **Eco Optimizer**.
6. Unselect one of the smells you noticed earlier.
7. Navigate back to the `sample.py` file.
8. Detect the smells again. Is the smell you unselected still there?

Participant Data

The following links point to the data collected from each participant:

Participant 1

Participant 2

Participant 3

Participant 4

Participant 5

Pre-Test Survey Data

The following link points to a CSV file containing the pre-survey data:

Click here to access the survey results CSV file.

Post-Test Survey Data

The following link points to a CSV file containing the post-survey data:

Click here to access the survey results CSV file.

Appendix — Reflection

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

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. Which parts of this document stemmed from speaking to your client(s) or a proxy (e.g. your peers)? Which ones were not, and why?

4. In what ways was the Verification and Validation (VnV) Plan different from the activities that were actually conducted for VnV? If there were differences, what changes required the modification in the plan? Why did these changes occur? Would you be able to anticipate these changes in future projects? If there weren't any differences, how was your team able to clearly predict a feasible amount of effort and the right tasks needed to build the evidence that demonstrates the required quality? (It is expected that most teams will have had to deviate from their original VnV Plan.)

Mya Hussain

- *What went well while writing this deliverable?*

  One of the most rewarding parts of completing this report was writing and compiling the benchmarking and performance analysis. Seeing the data come to life through plots and visualizations was very satisfying as we could see the underlying patterns we knew existed in our code in a visual format. It also outted everyones performance on their corresponding refactorers which was cool. Overall, the whole process of turning raw data into meaningful insights was really fulfilling. It felt like I was uncovering useful information that could really help improve the tool, which made the effort feel worthwhile

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

  The biggest pain point for me was definitely the sheer amount of unit testing that had to be done before even starting the report. Writing all those tests and making sure everything worked as expected was a lot of legwork, it felt like I was stuck in an endless loop of running tests, fixing bugs, and then running more tests. It was necessary but not the most exciting part of the process. The tricky part was making sure the report actually reflected all that effort. As we spent hours testing, and finding bugs, and fixing them, so the tool is a lot better, but logging all of those fixes without 1. sounding like the tool was broken to start and 2. overselling all the trivial tests we felt like we had to do to achive coverage, was a challenge.

Sevhena Walker

- *What went well while writing this deliverable?*

  A big win was how much of our work naturally fed into the report. Since we had already been refining our verification and validation (V&V) process throughout development, we weren't starting from scratch, we just had to document what we had done. Having clear test cases in place made it easier to describe our approach and results, rather than writing purely in the abstract. Another positive was that our understanding of the system had improved significantly by this point, so explaining our reasoning behind certain tests felt more natural.

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

  One challenge was finalizing our tests while also writing about them. Since we were still adjusting some test cases, we had to ensure that any changes were reflected correctly in the report, which meant some back-and-forth edits. Another issue was balancing detail; some sections needed more explanation than expected, while others felt overly technical. We resolved this by reviewing each section with fresh eyes and making sure we explained things clearly without unnecessary complexity. Time was also a factor, as wrapping up both testing and documentation at the same time was a bit hectic. We managed by setting smaller milestones to keep things on track and making sure to check in regularly to avoid last-minute rushes.

Ayushi Amin

- *What went well while writing this deliverable?*

  One of the best parts of working on this deliverable was how well my team collaborated. We had a clear understanding of what needed to be covered, which made it easier to

organize our thoughts and avoid unnecessary back-and-forth. Writing about our unit tests was also pretty smooth since we had already put a lot of effort into designing them in the vnv-plan. It was satisfying to document the thought process behind them, especially since they played a big role in making sure the tool was accurate and functioned correctly. Another thing I really enjoyed was usability testing. It was fun to see how others interacted with our tool and to get real feedback on what worked and what did not. Seeing users struggle with certain parts that we thought were intuitive was interesting to find out, but it also made the process more rewarding because we could make meaningful improvements.

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

  One pain point I experienced was structering the unit tests report and tracing back to the VnV plan tests. This is because The samples we had were really all over the place and not consistent at all. It was difficult to know what information was required for certian portions when most of the samples did not cover some portions. Also tracing back to VnV Plan tests, I realized that that some tests were not feasible and it would make no sense to do them for this project. Not entirely sure what we were thinking when we wrote them. So we decided to modify our VnV plan to be more realistic with the time frame we have and since a lot was changed in the scope of this project, we removed certain tests to better suit our current project.

Nivetha Kuruparan

- *What went well while writing this deliverable?*

  One of the things that went well while working on this deliverable was our ability to catch a significant number of bugs and edge cases during testing. Through extensive unit and integration testing, we identified multiple issues related to multi-file refactoring, detection accuracy, and performance optimization. This allowed us to refine our detection and refactoring mechanisms, making them more reliable and robust.

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

  One of the biggest challenges we faced was the overwhelming number of tests outlined in the original V&V Plan. While comprehensive, implementing every test and writing detailed reports for each became highly time-consuming and impractical. As a result, we had to carefully trim down and consolidate tests to focus on the most critical functionalities while still maintaining full coverage of our system requirements. This process involved combining similar tests and prioritizing cases that had the most significant impact on correctness, usability, and performance. While this required careful review and restructuring, it ultimately streamlined the validation process and improved efficiency in writing the report.

Tanveer

- *What went well while writing this deliverable?*

  The fun part was validating different requirements that we had defined in the VnV Plan against our tool. I saw that some of them were too ambitious versus others could have more points added for the verification. Overall, it was fun mapping non functional requirements against the features of the tool. At the end of it, I was able to deduce which NFR maps to a certain feature of the tool.

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

  Writing unit tests turned out to be harder than actual implementation because (1) not only did I come across bugs when testing but also (2) mocking dependencies such as `vscode.workspace` for our plugin was definitely a learning curve. It is important to mentiont that I don't believe that the course and capstone would have been the same as testing, the team was testing left and right to get the maximum coverage. To resolve the learning curve I referred to multiple tutorials online and eventually the process became getting rid of the syntax errors or bugs in the unit test implementation so that the tests could pass.

Group

- *Which parts of this document stemmed from speaking to your client(s) or a proxy (e.g. your peers)? Which ones were not, and why?*

  Parts of this document stemmed from speaking to users who acted as proxies for clients. Specifically:

  - **Usability Testing Findings:** The document details usability testing conducted with student developers, who served as proxies for real-world users. Their feedback on sidebar visibility, refactoring speed, UI clarity, and energy savings feedback directly influenced the report.
  - **Methodology and Results:** The task completion rates, user satisfaction scores, and qualitative insights were derived from these interactions, making them user-driven.
  - **Non-functional Requirements:** This is based on client as some requirements like look and feel is evaluated by client and usability testers since they will be the ones using the application.

  Parts of the document that did not stem from client or proxy interactions include:

  - **Functional Requirements Evaluations:** These sections reference predefined specifications/industry standards rather than direct client input.

  - **Implementation and Technical Explanations:** These were formulated based on the development team's decisions, software documentation, and prior knowledge rather than external feedback.

- *In what ways was the Verification and Validation (VnV) Plan different from the activities that were actually conducted for VnV? If there were differences, what changes required the modification in the plan? Why did these changes occur? Would you be able to anticipate these changes in future projects? If there weren't any differences, how was your team able to clearly predict a feasible amount of effort and the right tasks needed to build the evidence that demonstrates the required quality? (It is expected that most teams will have had to deviate from their original VnV Plan.)*

  There were definitely some differences between what we assumed would happen during the VnV Plan and the results we actually got during testing. For example, the plan initially assumed that energy measurement times would vary significantly with file size, but the testing revealed that they were actually decently consistent. This meant we had to adjust our focus in the report to highlight the fixed overhead of energy measurement rather than exploring variability.

  For the most part, though, all of the unit testing we planned in the VnV Plan was written out per spec, and the code was fixed until all of them passed. This rigorous testing process actually caught a lot of bugs and edge cases that we hadn't fully anticipated in the plan. For instance, some refactoring operations worked fine on smaller files but broke on larger ones. Testing also revealed edge cases, like how the tool handled files with **multiline whitespace**, **nested structures**, **degenerate/trivial input** (e.g., empty files or files with a single line), and **wrong input** (e.g., malformed code or unsupported syntax). These cases weren't explicitly called out in the original plan, but they became a big part of the testing process once we realized how critical they were to the tool's reliability.

  For example:

  - **Multiline whitespace:** The tool initially struggled with files that had excessive or irregular whitespace, which caused false positives in code smell detection. We had to update the detection logic to handle these cases gracefully.
  - **Nested structures:** Deeply nested code (e.g., loops within loops or functions within functions) exposed performance bottlenecks and sometimes caused the tool to crash. This led to optimizations in the refactoring algorithms.
  - **Degenerate/trivial input:** Empty files or files with minimal content revealed that some refactoring operations weren't properly handling edge cases, so we added checks to ensure the tool behaved correctly in these scenarios.
  - **Wrong input:** Malformed or unsupported code caused unexpected errors, so we improved error handling and added clearer feedback for users.

  Fixing these issues required additional effort, but it ultimately made the tool more robust and user-friendly.

These changes happened because testing revealed patterns in the data and uncovered bugs that weren't obvious during the planning phase. The bugs and edge cases we found during testing forced us to revisit parts of the code and make improvements we hadn't planned for initially.

Some of these changes could be anticipated in future projects with more thorough initial testing. If i could do it again I'd build more flexibility into the VnV Plan to account for unexpected results and allocate extra time for debugging and edge-case testing. I'd also include a broader range of test cases (e.g., multiline whitespace, wrong input) in the initial plan to catch these issues sooner.