LLM Driven Unit Test Case Generation Using Agentic AI

Baskaran S.[1], Pradeepta Mishara[2], Rashmi Agarwal[3]

[1,3]REVA Academy for Corporate Excellence, REVA University, Bangalore, India

[2]AI Innovation, Beghou Consulting, Bangalore, India

E-mail: [1]baskaran.ai06@race.reva.edu.in, [2]pradeeptamishra@race.reva.edu.in, [3]rashmi.agarwal@reva.edu.in

Abstract

Unit testing plays a crucial role in application software development by validating module functionality in isolation before system integration. Manually writing and reviewing unit test cases is time-consuming and defect-prone. Complex logic and boundary conditions are not tested thoroughly, leading to higher rework costs. Automated test generation using Large Language Models (LLMs) reduces development effort but faces challenges such as ensuring meaningful test coverage, handling invalid inputs, and addressing missing imports. This study aims to leverage LLMs in combination with the Autogen Agentic AI framework to generate high-quality Python unit tests by effectively prompting them, fixing failed test cases, validating them through test execution, analyzing results, and improving code coverage and mutation score. For experiments conducted on the Insurance Management Application, branch coverage improved from 98% to 99%, and the mutation score improved from 83.9% to 95.8%. The proposed approach significantly reduces manual effort while improving test suite effectiveness and software quality.

Keywords: Unit Testing, Large Language Models, Agentic AI, Test Automation, AI-Driven Testing, Test Case Effectiveness, Code Coverage, Mutation Testing.

1. Introduction

During application development, individual modules must undergo thorough testing in isolation before being integrated into the system. Unit test execution helps to verify that all

DOI: https://doi.org/10.36548/jucct.2025.4.003
Received: 02.10.2025, received in revised form:03.11.2025, accepted: 18.11.2025, published: 28.11.2025

boundary conditions and module functions are working correctly. However, manually creating unit test cases is time-consuming and error-prone, increasing review time and rework costs.

Large language models, such as OpenAI's GPT, Google's Gemini, and Meta's Llama, represent a new generation of algorithms that can synthesize unit test cases for code modules. As such, this new generation can greatly reduce development effort. Some of the challenges in test case generation involve meaningful test coverage, invalid inputs, and missing imports. Jain and Le Goues present feedback-driven and agentic approaches to improve test suite quality and reduce human effort using large language models [1]. Writing unit test cases by hand is slow, laborious, and error-prone, resulting in incomplete test coverage and defects that are more expensive to fix later. Although LLM-based automation is in its heyday, it tends to fail with complex scenarios where multiple classes and invalid inputs may be involved, with no guarantee of achieving comprehensive coverage. There is, therefore, an urgent need for more innovative methods of automated testing capable of reducing manual effort while guaranteeing thorough and reliable quality. This research aims to apply the Agentic AI AutoGen framework with LLMs toward the automation of unit test case generation in Python programs via effective prompting. Besides generating test cases, this study also investigates refining test cases to improve branch coverage and mutation scores, such as MuTAP [2] proposed by Dakhel et al., in order to highlight defects in test logic and further harden the robustness of tests.

The rest of the paper is organized as follows: Section 2 reviews the related research and literature; Section 3 explains the proposed methodology and system design, while Section 4 describes the implementation; Section 5 discusses the analysis and results, and finally Section 6 presents the conclusion and discusses some possible future research directions.

2. Literature Review

Large language models generate unit test cases that overcome traditional challenges such as test case thoroughness and code coverage. Wang et al. [3] proposed a method for slicing complex methods into smaller units and generating comprehensive unit test cases using large language models. Zhang et al. [4] introduced property-based retrieval mechanisms within Retrieval-Augmented Generation (RAG) frameworks to enhance the relevance of generated tests. Meta's industrial-scale implementation with TestGen-LLM by Alshahwan et al. [5] demonstrated the practical feasibility of LLM-based testing to improve human-written test cases, eliminate hallucinations, and attain superior coverage metrics in production environments.

Chen et al. [6] introduced ChatUniTest, which combines adaptive focal context mechanisms with generation, validation, and repair processes to ensure the quality of unit test cases. Pizzorno and Berger [7] developed CoverUp, a coverage-guided approach to iteratively improve test case generation using real-time code coverage feedback. Recent advances by Gu et al. [8] have introduced hybrid program analysis techniques that combine static control flow analysis with dynamic code coverage to improve unit test case generation for untested execution paths

Building on these advancements, Ryan et al. [9] introduced Code-Aware Prompting, a coverage-guided approach to regression test generation. This method uses large language models to enhance test coverage and improve defect detection. Their work demonstrates how combining code understanding with real-time coverage feedback can make LLM-based test generation more effective and reliable in practical development environments.

In addition to these core contributions, several studies further expand the scope of LLM-assisted testing. Storhaug and Li [10] showed that parameter-efficient fine-tuning improves the accuracy and stability of LLM-generated tests. Pan et al. [11] introduced ASTER, a multilingual, NL-driven testing framework supporting various programming ecosystems. Zhang et al. [12] proposed CITYWALK, which uses project-level dependency structures to improve C++ unit test accuracy. Bhatia et al. [13] compared generative AI test-generation tools, while Zhong et al. [14] and Bayri & Demirel [15] demonstrated the effectiveness of LLMs in enhancing defect detection and modernizing testing workflows.

Despite significant advancements in utilizing large language models for unit test generation, current research does not present a comprehensive end-to-end system that seamlessly integrates code chunking, prompting, test creation, execution, coverage analysis, mutation feedback, and automated repair within a unified pipeline. Many existing approaches also encounter consistency challenges due to their reliance on probabilistic outputs from large language models, which lack adequate validation mechanisms. Moreover, the potential of mutation testing to verify tests generated by large language models remains underexplored.

A multi-agent, feedback-driven methodology has been proposed to address these limitations. The methodology provides an integrated framework intended to overcome the highlighted deficiencies.

3. Methodology and System Design

The CRISP-DM framework has been chosen for its iterative and structured approach, fitting the requirements of automated unit test case generation workflows. Indeed, all its different phases map directly to the various stages of Large Language Model-based testing: Business Understanding corresponds to identifying target code modules; Data Preparation involves source code extraction and chunking; Modeling encompasses generating test by creating appropriate prompts; evaluation covers test execution and coverage analysis; and Deployment involves validated tests execution via Continuous Integration/Continuous Deployment pipelines. A crucial reason for CRISP-DM's effectiveness is its inherently cyclical nature, allowing for continuous improvement of this process by incorporating feedback from test results and coverage gaps.

The system is a network of interconnected modules: automating the construction of test cases, execution, measurement of coverage, and improvement of the mutation score. The key components involved in the system are a Streamlit interface, a test-generation agent, chunking and prompt modules, a test-execution agent, a coverage agent, a mutation-testing agent, an auto-fix agent, and tools for managing mutations and files. Figure 1 illustrates a detailed diagram of this system.

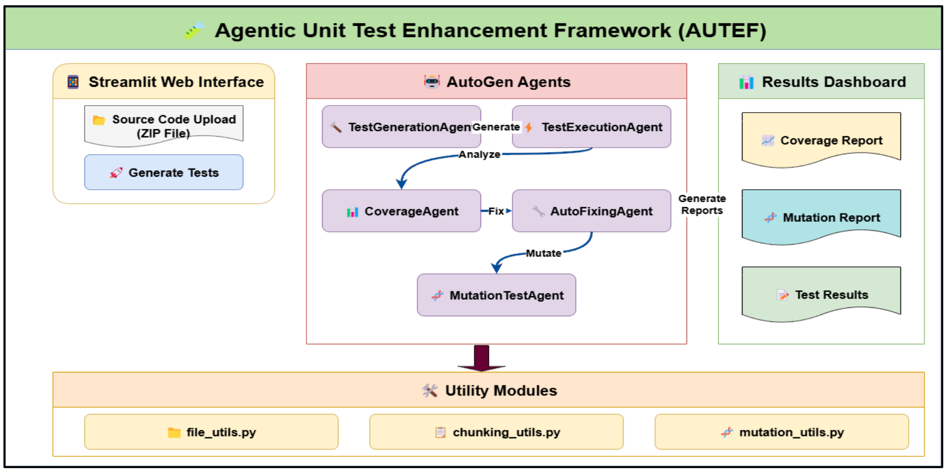


Figure 1. High-Level Design Diagram

Unlike a single-agent workflow, it is shared among multiple agents in the multi-agent architecture: the generation of unit test cases, execution, code coverage analysis, mutation score evaluation, and fixing unit test case failures. This clear division reduces the cognitive load on individual agents, enables iterative improvements, and enhances coverage and mutation scores through coordinated inter-agent effort processes.

Figure 2 depicts the overall workflow of the automated unit test-generation pipeline. The approach starts with source code analysis and chunks the code into meaningful pieces. The LLM generates some initial unit test cases based on suitable prompts and input code chunks. The generated tests are executed and their results are analysed via code coverage assessment, which identifies the branches that remain untested. In cases where coverage gaps are identified, the model is provided with a suitable prompt to generate new tests that increase coverage. Failing or incomplete tests are automatically fixed through an iterative fixing process.

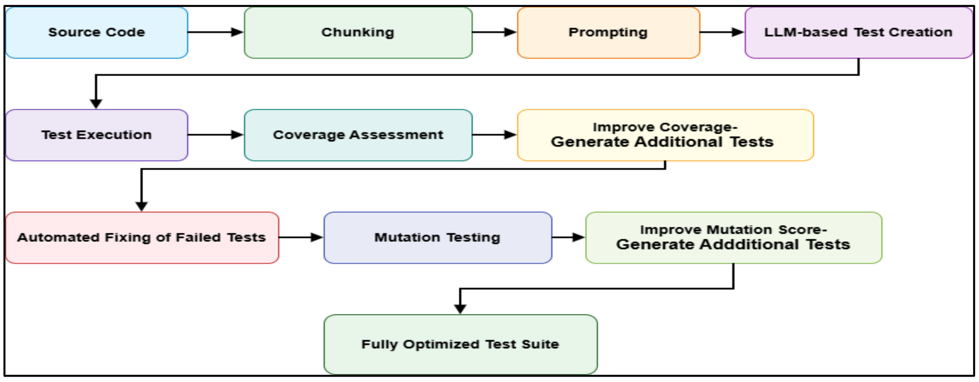


Figure 2. LLM-Based Test Generation Workflow

The reason mutation testing has been selected as the main validation method is that it provides a direct and quantitative measure of the fault-detection ability of a test suite. Property-based testing, on the other hand, requires domain-specific invariants, which in general are lacking for business-oriented applications. The proposed approach based on mutation testing represents a language-agnostic, specification-independent assessment method. In this respect, it is much better suited to LLM-generated tests. Mutation testing measures the capability of tests to detect changes in program behavior. Stronger test cases are developed to kill the surviving mutants and increase the mutation score. This create-run-analyze-fix cycle results in a fully optimized and reliable test suite. While the natural tendency of an LLM is to be non-deterministic, this framework fosters deterministic output by using fixed decoding parameters, such as a temperature of 0; it enforces strict formats for output, controls test structure by using templates, and validates generated tests by execution and performing mutation analysis. Only tests that always pass, over multiple runs, are retained, ensuring stability in the final test suite.

The following sections explain the key components of the proposed framework.

3.1 Unit Test Case Generation Agent

The Unit Test Generation Agent uses prompts and the source code to auto-generate unit test cases for all functions and classes. Cases are also generated for testing boundary conditions, positive scenarios, and error conditions.

3.2 Unit Test Execution Agent

The unit test agent executes unit tests and returns test results in both HTML and JSON formats. The test report includes test cases, test execution status (success/failure), and reasons for failure to quickly help developers fix the defects. These test executions can also be integrated with CI/CD pipelines to ensure automated testing.

3.3 Coverage Agent

The coverage agent measures the test coverage for lines, functions, and branches, showing a detailed coverage report that includes metrics like line coverage and branch coverage. Areas where improvement is required are highlighted in the report as well. It also generates additional test cases to improve the coverage of areas that require improvement.

3.4 Auto Fixing Agent

It auto-detects test failures and fixes them on its own, reducing developers' efforts by requiring minimal manual intervention.

3.5 Mutation Testing Agent

The mutation testing agent introduces small mutations in the codebase and verifies whether the current test suite detects them. It provides a comprehensive report, showing the number of mutants killed, survived, and incompetent, along with the respective mutation score. The agent presents reports in HTML and YAML formats. It further enhances test effectiveness by adding new test cases based on the surviving mutants.

3.6 Utility Modules

The Chunking Utils module splits large source code files into smaller, manageable chunks for LLM processing. This ensures each chunk is within max_chunk_size, which is by default 512 characters, and maintains context during unit test generation. File Utils provides helper functionality for managing files throughout the test generation process. These include listing, reading, and writing files; creating necessary directories for processing; unzipping ZIP files uploaded for batch processing by a user; and associating source files with their corresponding test files. Mutation Utils assists in creating and applying mutations to the source code to evaluate unit test effectiveness. This supports multiple types of mutations and provides visual reports to help developers improve the mutation score of unit test results.

3.7 Template Prompts

The Template Prompts module includes prompts for unit test case generation, measuring and improving branch coverage, fixing test case failures, and measuring and improving mutation scores. The test case generation prompt creates comprehensive unit test cases for a given chunk of code, including necessary imports to ensure corner cases are covered. The branch coverage prompt identifies missing branches or execution paths in your test cases and generates test cases for these missing branches, helping to improve the branch coverage of your test suite. The test case failure fix prompt is used when a test case fails. Its primary objective is to restore the broken unit test based on the functionality of the source code and the reasons for the associated failure of a test. By analyzing both the source code and the failing test function, the prompt applies fixes directly to the test cases. The mutation coverage improvement prompt enhances the mutation score by generating additional test cases around any surviving cases.

4. Implementation

This work presents a case study of an Insurance Management System with numerous modules and complex business logic to demonstrate our approach. The AutoGen Agentic AI framework automatically generates unit tests, executes the generations, and refines them using coverage, mutation, and auto-fixing agents. Structured prompts direct the LLM to generate accurate and meaningful test cases while automated analysis identifies coverage gaps and failing tests for resolution. This section discusses the system modules, the AutoGen framework, prompt templates, test case generation, execution, and coverage enhancement.

4.1 Insurance Management System

The system consists of about 75 lines of Python code, divided into four classes and 12 methods. The architecture includes API handlers, business logic modules, repository components, and persistence utilities. This forms a nice multi-layer environment where unit test generation, improvement of branch coverage, and mutation testing can be realistically assessed. Its various modules collectively handle policy management, input validation, data storage, and enforcement of business rules, making the application quite suitable for validating both functional and structural aspects. API Layer - policy_handler: provides RESTful operations and input validation on incoming requests to add policies. Service Layer - policy_service: encapsulates business logic for special conditions, rules, validation of inputs for policies, and cancellation logic. Repository Layer - policy_repository: responsible for accessing and managing policy data in memory or through a storage abstraction. The Persistence Layer - database: provides a thin interface that supports insert, delete, and fetch operations along with error handling corresponding to duplicate entry, invalid ID, and record not found.

4.2 AutoGen Agentic AI Framework

The unit test cases are generated using the agentic framework of AutoGen. It performs the enhancement through different agents, which include coverage, mutation, and auto-fixing. The test execution agent executes the generated test cases and provides test execution reports.

4.3 Prompt Templates

The framework uses targeted prompt engineering with well-designed prompt templates including unittest structures, appropriate imports, assertions, and exception handling. Rules being path-aware allow for proper relative imports. Error feedback prompts support the auto-correction of failing unit tests. Mutation-aware prompts are used in creating tests that can reach the surviving mutants. For that, the Prompt Template Module crafts clear, factual prompts to guide the LLM in generating, refining, and optimizing unit test cases. It selects an appropriate prompt based on the testing objective that may range from creating initial tests to improving branch coverage or enhancing robustness through mutation testing. The module ensures that each prompt is complete, accurate, and tuned for the specific test-generation task by extracting essential details like code snippets, file paths, function names, and existing tests.

Depending on the goal, the module uses a specific prompt builder. It uses get_prompt() to create a complete test. It uses generate_branch_coverage_prompt()to target unreachable branches. It uses generate_mutation_prompt() for mutation testing. Metadata such as discovered branches, test failure logs, or mutation descriptions are also included in the prompts in this module. Prompts are formatted to follow the rules: keeping imports, mocking, handling exceptions, and ensuring syntax correctness. After the language model produces or updates code, this module processes it and adds it to the right test files. This ensures tests are accurate and complete, matching the goals.

4.4 TestCase Generation

The test generation agent will utilize Open AI's language model via the Agentic AI framework to generate unit tests from code using the autogen feature. It supports large codebases by chunking the files into pieces. Each piece is fed into the model along with a prompt, and the model produces a test script. Following test generation, system tokens and other unnecessary details will be removed from the output, so only the cleaned test files are saved to the given test folders and are ready to run.

4.5 TestCase Execution

It automatically detects and executes unit tests. Feedback will be provided in JSON and HTML formats. Setup files and environment variables are loaded by the agent. The agent sets up the configuration with an inserted OpenAI API key. The agent looks for test files, typically named test_*.py, and executes them all. Tests are executed using Python's unittest framework. Results are marked as PASS, FAIL, or ERROR. Next, it creates a JSON file containing timestamps for tracking. It also generates an HTML report styled with Pandas and presents the results clearly.

4.6 Coverage Improvement

The Coverage Agent measures and improves code coverage. It helps ensure that unit tests have thoroughly covered all parts of the code. The agent runs the existing unit tests and produces a coverage report by using the module coverage.py. This report also indicates which parts of the code have not yet been tested. The agent analyzes the report in order to identify untested code sections that need more testing. Next, the agent designs new test methods using an appropriate prompt provided to the Large Language Model in order to fill these gaps and therefore increase the coverage of untested code. The prompt would include coverage data, missing code sections and the current source and test files. These are then added to the corresponding test classes. The agent again reruns the unit tests and issues an updated report on coverage analysis. The revised coverage results are shown in the terminal and HTML format. This helps the developers see the improvements made and increases confidence that the code is well tested.

4.7 Fix Failing Unit Tests

The AutoFixing Agent minimizes the manual effort required to find and fix failing unit tests. It runs all the unit tests and saves the results in an easy-to-parse JSON format. The agent then parses these results to identify any tests that failed or threw errors. The agent locates the corresponding test files in the project by class name. Once the relevant test methods have been identified, the agent uses rules to fix simple problems. For other failures, it retrieves the related source code. It sends a context-aware prompt to the LLM with the request to propose a fixed version of the test method. It replaces the broken test method with a new one in the test file and reruns the tests to verify the fix. Therefore, this automated process supports the quick resolution of failing tests, while maintaining consistency in the test suite.

4.8 Mutation Coverage Improvement

The Mutation Testing Agent automates mutation testing. It checks the unit tests and finds surviving mutants. Then, it creates new tests to catch faults not currently tested. The agent finds the source and test files in the project folder. It uses mut.py to generate mutants by making minor changes, such as changing operators or flipping conditions. The agent runs the current tests against these mutants. It sees which mutants survive, revealing gaps in the tests. Details about each surviving mutant, such as the type of mutation, lines affected, and surrounding code, are used by the agent to compose a prompt for the Large Language Model. Based on this prompt, it generates new or improved test cases to identify the surviving mutants. These are added to the test classes of the code. The agent reruns the mutation tests, after which the results show whether test coverage is sufficient.

5. Test Results and Analysis

This section presents the test results along with the analysis of improvements in code coverage and mutation scores.

5.1 Test Results

The test cases were executed using Python's unittest framework, and the results were analyzed. Initial test execution showed forty-two passing tests. Table 1 presents a sample test case result of the HTML output from the test execution. From the execution results, all expected outputs matched the actual results, indicating correct functionality. The application correctly raised ValueError and TypeError for invalid input types. The framework successfully captured and validated exception handling for erroneous test cases.

Table 1. Unit Test Case Results

Test Case	Status	Reason
InsuranceApp_Modified.app.api.test_policy_handler_0.TestPolicyHandler.test_add_policy_high_id_low_coverage_holder_a	PASS	None
InsuranceApp_Modified.app.api.test_policy_handler_0.TestPolicyHandler.test_add_policy_high_id_valid_coverage_holder_a	PASS	None
InsuranceApp_Modified.app.api.test_policy_handler_0.TestPolicyHandler.test_add_policy_high_id_valid_coverage_holder_b	PASS	None
InsuranceApp_Modified.app.api.test_policy_handler_0.TestPolicyHandler.test_add_policy_invalid_coverage_amount_negative	PASS	None
InsuranceApp_Modified.app.api.test_policy_handler_0.TestPolicyHandler.test_add_policy_invalid_coverage_amount_zero	PASS	None

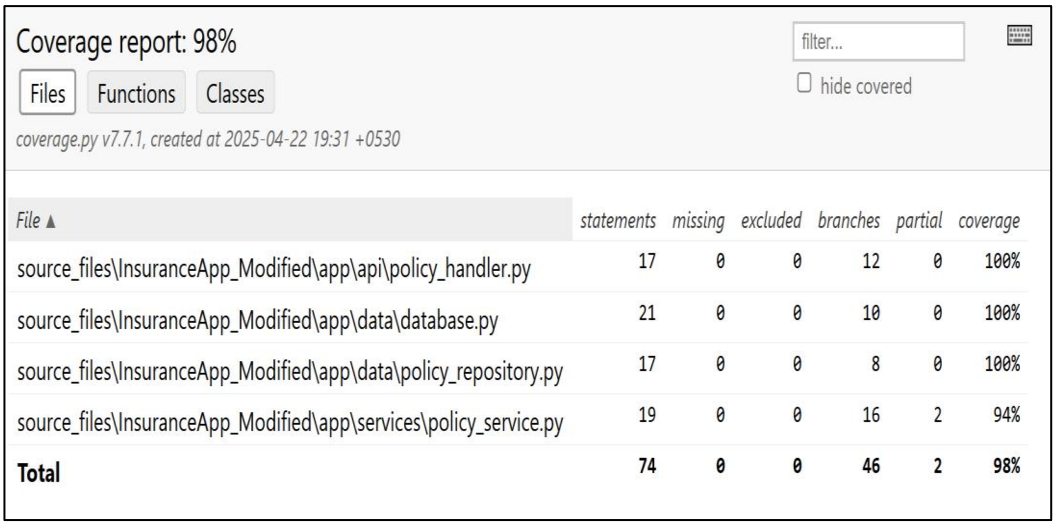
Subsequent improvements focused on increasing test coverage, particularly for modules with conditional logic not fully exercised by the original tests. After enhancements, the number of test cases rose to forty-four, and all tests passed successfully. Table 2 summarizes the number of test cases executed before and after the enhancements.

Table 2. Unit Test Results Analysis Results

Test Class	Initial Test Cases	Enhanced Test Cases
Test Policy Handler	11	11
Test Database	10	10
Test Policy Service	13	15
Test Policy Repository	8	8
Total	42	44

Adding two test cases to the TestPolicyService class strengthens the validation of service-level logic, specifically addressing previously uncovered branches and conditions.

5.2 Code Coverage Analysis



Coverage report: 98%

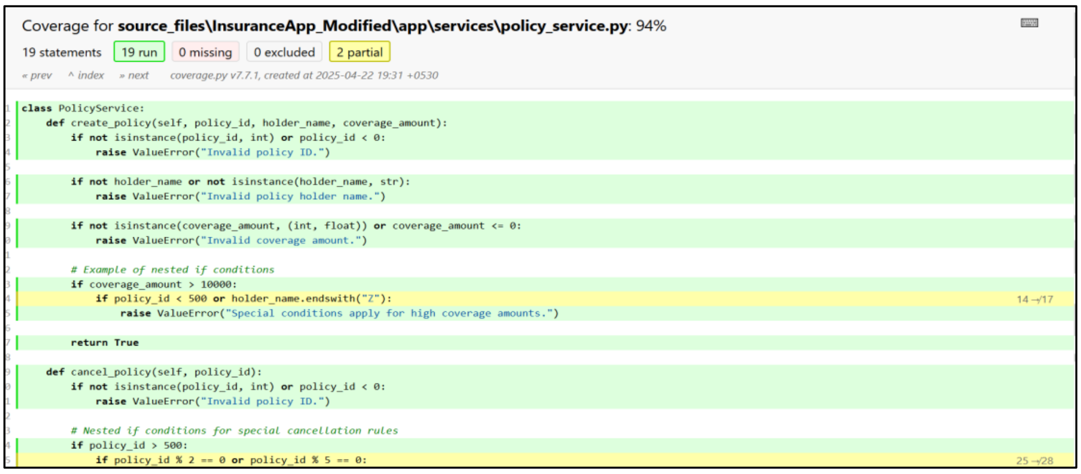
coverage.py v7.7.1, created at 2025-04-22 19:31 +0530

File ▲	statements	missing	excluded	branches	partial	coverage
source_files\InsuranceApp_Modified\app\api\policy_handler.py	17	0	0	12	0	100%
source_files\InsuranceApp_Modified\app\data\database.py	21	0	0	10	0	100%
source_files\InsuranceApp_Modified\app\data\policy_repository.py	17	0	0	8	0	100%
source_files\InsuranceApp_Modified\app\services\policy_service.py	19	0	0	16	2	94%
Total	**74**	**0**	**0**	**46**	**2**	**98%**

Figure 3. Initial Code Coverage(Screenshot)

The initial code coverage analysis showed an overall branch coverage of 98%, indicating good but incomplete test coverage across all source files. Figure 3 shows the code coverage for the individual modules.

Additional analysis identified that some branches in the *PolicyService* class were not adequately tested, as shown in Figure 4. This figure highlights the uncovered branches at the methods create_policy and cancel_policy.


Coverage for **source_files\InsuranceApp_Modified\app\services\policy_service.py**: 94%

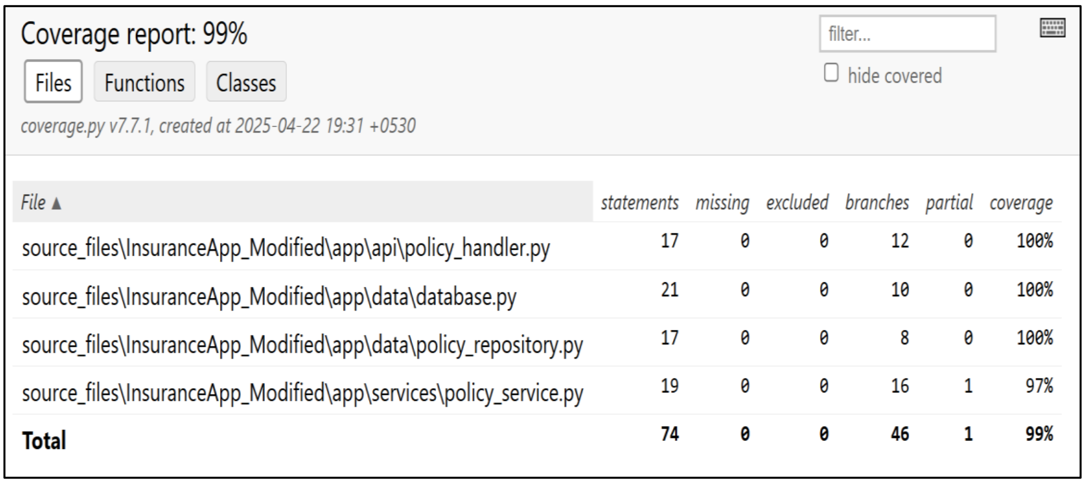
19 statements | 19 run | 0 missing | 0 excluded | 2 partial

« prev ^ index » next coverage.py v7.7.1, created at 2025-04-22 19:31 +0530

```
class PolicyService:
    def create_policy(self, policy_id, holder_name, coverage_amount):
        if not isinstance(policy_id, int) or policy_id < 0:
            raise ValueError("Invalid policy ID.")

        if not holder_name or not isinstance(holder_name, str):
            raise ValueError("Invalid policy holder name.")

        if not isinstance(coverage_amount, (int, float)) or coverage_amount <= 0:
            raise ValueError("Invalid coverage amount.")

        # Example of nested if conditions
        if coverage_amount > 10000:
            if policy_id < 500 or holder_name.endswith("Z"):                                14 ↛ 17
                raise ValueError("Special conditions apply for high coverage amounts.")

        return True

    def cancel_policy(self, policy_id):
        if not isinstance(policy_id, int) or policy_id < 0:
            raise ValueError("Invalid policy ID.")

        # Nested if conditions for special cancellation rules
        if policy_id > 500:
            if policy_id % 2 == 0 or policy_id % 5 == 0:                                    25 ↛ 28
```

Figure 4. Missing Branches -Policy Service -Initial (Screenshot)

Following the test enhancement phase, the improved coverage report showed a measurable increase in code coverage, with total coverage reaching 99%. The updated module-level coverage distribution is presented in Figure 5.

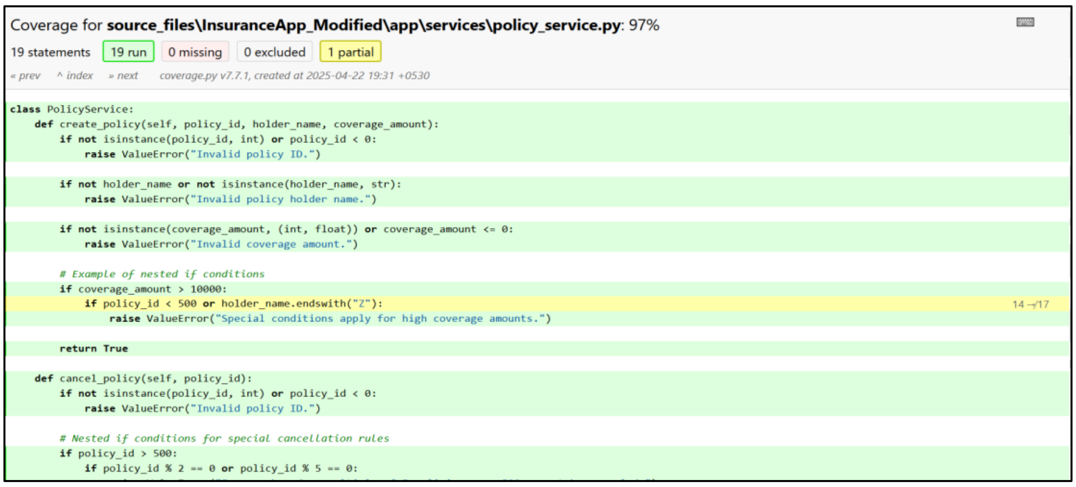

Coverage report: 99%

coverage.py v7.7.1, created at 2025-04-22 19:31 +0530

File ▲	statements	missing	excluded	branches	partial	coverage
source_files\InsuranceApp_Modified\app\api\policy_handler.py	17	0	0	12	0	100%
source_files\InsuranceApp_Modified\app\data\database.py	21	0	0	10	0	100%
source_files\InsuranceApp_Modified\app\data\policy_repository.py	17	0	0	8	0	100%
source_files\InsuranceApp_Modified\app\services\policy_service.py	19	0	0	16	1	97%
Total	**74**	**0**	**0**	**46**	**1**	**99%**

Figure 5. Improved Code Coverage (Screenshot)

The additional test cases' impact on the *PolicyService* class is evident in the reduction of missing branches, as shown in Figure 6. The enhanced test cases successfully targeted previously untested control paths, improving the verification of decision-making logic within the service layer.



Coverage for **source_files\InsuranceApp_Modified\app\services\policy_service.py**: 97%

19 statements | 19 run | 0 missing | 0 excluded | 1 partial

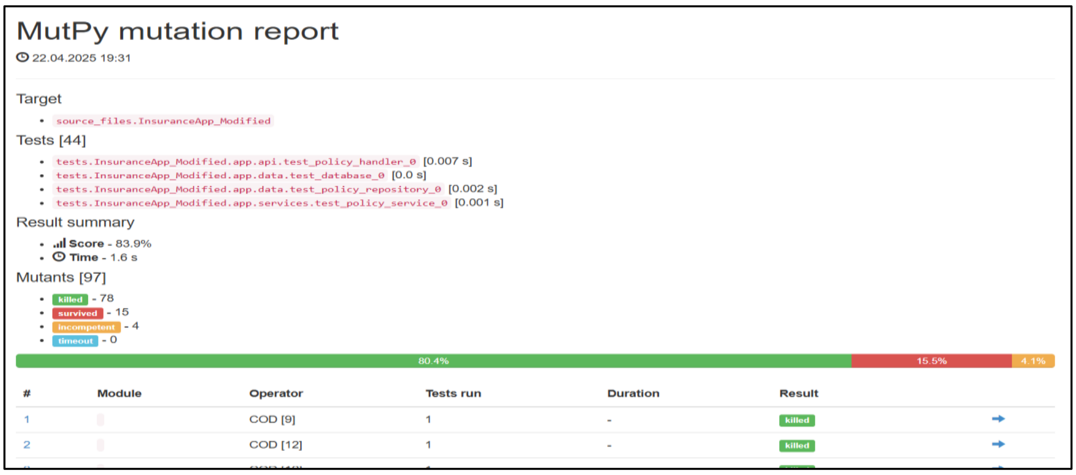
« prev ^ index » next coverage.py v7.7.1, created at 2025-04-22 19:31 +0530

```python
class PolicyService:
    def create_policy(self, policy_id, holder_name, coverage_amount):
        if not isinstance(policy_id, int) or policy_id < 0:
            raise ValueError("Invalid policy ID.")

        if not holder_name or not isinstance(holder_name, str):
            raise ValueError("Invalid policy holder name.")

        if not isinstance(coverage_amount, (int, float)) or coverage_amount <= 0:
            raise ValueError("Invalid coverage amount.")

        # Example of nested if conditions
        if coverage_amount > 10000:
            if policy_id < 500 or holder_name.endswith("Z"):                          14 ↛ 17
                raise ValueError("Special conditions apply for high coverage amounts.")

        return True

    def cancel_policy(self, policy_id):
        if not isinstance(policy_id, int) or policy_id < 0:
            raise ValueError("Invalid policy ID.")

        # Nested if conditions for special cancellation rules
        if policy_id > 500:
            if policy_id % 2 == 0 or policy_id % 5 == 0:
```

Figure 6. Missing Branches-Policy Service-Updated (Screenshot)

5.3 Mutation Score Analysis

The initial mutation testing achieved a score of 83.9%, showing a strong test suite. Of the ninety-seven mutants introduced, seventy-eight were killed, and fifteen survived, revealing logic paths not entirely validated by existing tests; four were marked as incompetent. After adding targeted test cases to cover gaps, analyzing surviving mutants, and updating source files, the mutation tests were rerun. This resulted in the death of eleven more mutants. No incompetent mutants remained, indicating better test accuracy, and the mutation score increased from 83.9% to 95.9%. Figure 7 and Figure 8 provide the mutation report for the initial and improved mutation scores.



MutPy mutation report

22.04.2025 19:31

Target

- `source_files.InsuranceApp_Modified`

Tests [44]

- `tests.InsuranceApp_Modified.app.api.test_policy_handler_0` [0.007 s]
- `tests.InsuranceApp_Modified.app.data.test_database_0` [0.0 s]
- `tests.InsuranceApp_Modified.app.data.test_policy_repository_0` [0.002 s]
- `tests.InsuranceApp_Modified.app.services.test_policy_service_0` [0.001 s]

Result summary

- **Score** - 83.9%
- **Time** - 1.6 s

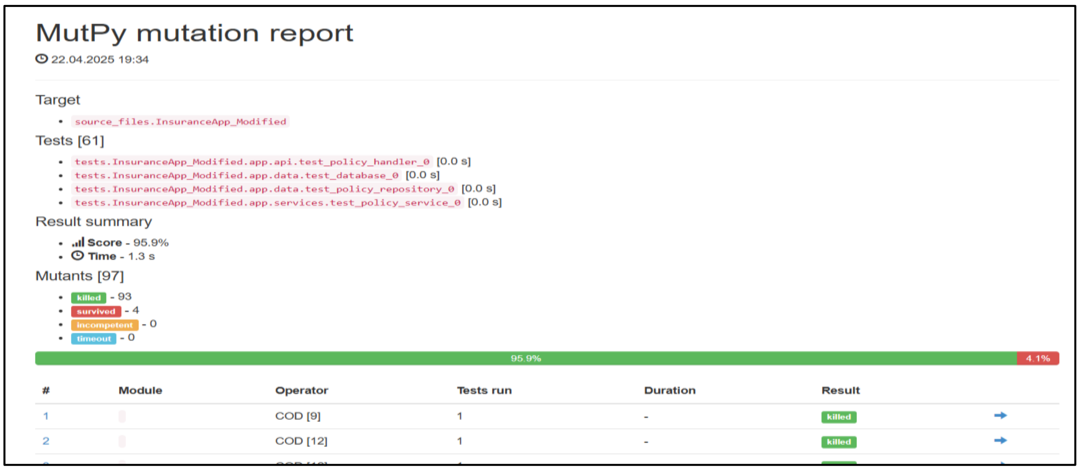
Mutants [97]

- killed - 78
- survived - 15
- incompetent - 4
- timeout - 0

80.4% | 15.5% | 4.1%

#	Module	Operator	Tests run	Duration	Result
1		COD [9]	1	-	killed
2		COD [12]	1	-	killed

Figure 7. Mutation Coverage-Initial (Screenshot)



MutPy mutation report

22.04.2025 19:34

Target

- `source_files.InsuranceApp_Modified`

Tests [61]

- `tests.InsuranceApp_Modified.app.api.test_policy_handler_0` [0.0 s]
- `tests.InsuranceApp_Modified.app.data.test_database_0` [0.0 s]
- `tests.InsuranceApp_Modified.app.data.test_policy_repository_0` [0.0 s]
- `tests.InsuranceApp_Modified.app.services.test_policy_service_0` [0.0 s]

Result summary

- **Score** - 95.9%
- **Time** - 1.3 s

Mutants [97]

- killed - 93
- survived - 4
- incompetent - 0
- timeout - 0

95.9% 4.1%

#	Module	Operator	Tests run	Duration	Result	
1		COD [9]	1	-	killed	→
2		COD [12]	1	-	killed	→

Figure 8. Mutation Coverage-Improved (Screenshot)

The proposed approach includes a comprehensive feedback loop that involves chunking, structured prompting, execution validation, coverage analysis, mutation-score improvement, and fixing failed tests. It develops more thorough, maintainable test suites, resulting in test quality metrics that match or exceed those of leading automated testing frameworks. Besides code coverage and mutation score, metrics such as assertion density, unique path exploration, oracle strength, flakiness rate across repeated runs, redundancy removal, and defect detection ability can be used to evaluate the effectiveness of LLM-generated test cases.

6. Conclusions and Future Work

Testing with large language models will improve test quality and enable earlier defect detection. Agentic AI can combine with LLMs to find many boundary test cases that might be missed in manual testing. These tools, when integrated with code coverage and mutation testing, provide objective feedback on test effectiveness. Greater coverage and mutation scores mean stronger test suites with earlier defect detection, less rework, and more reliable software. Clearly and concisely, reports are provided for faster resolution by the developer. There are several methods for enhancing LLM-based unit testing. Agentic AI may find test failures during test runs and then fix both the test cases and the application code. Additionally, it can improve mutation scores by running additional tests or by updating the code to implement required functionality changes. Testing more of the code requires clear prompt instructions and may necessitate trying different LLMs, such as those from OpenAI, DeepSeek, Cohere, or Google. Improving tools to support multiple programming languages, like Java, JavaScript, and C#, and integrating agents with development tools and build systems also enables teams to create, review, and continuously improve the test cases throughout software development. It also accelerates software development and enhances quality.

References

[1]. Jain, Kush, and Claire Le Goues. "TestForge: Feedback-Driven, Agentic Test Suite Generation." arXiv preprint arXiv:2503.14713 (2025).

[2]. Dakhel, Arghavan Moradi, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C. Desmarais. "Effective Test Generation using Pre-Trained Large Language Models and Mutation Testing." Information and Software Technology 171 (2024): 107468.

[3]. Wang, Zejun, Kaibo Liu, Ge Li, and Zhi Jin. "Hits: High-Coverage LLM-Based Unit Test Generation via Method Slicing." In Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, pp. 1258-1268. 2024.

[4]. Zhang, Zhe, Xingyu Liu, Yuanzhang Lin, Xiang Gao, Hailong Sun, and Yuan Yuan. "LLM-Based Unit Test Generation via Property Retrieval." arXiv preprint arXiv:2410.13542 (2024).

[5]. Alshahwan, Nadia, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. "Automated Unit Test Improvement using Large Language Models at Meta." In Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, 185-196. 2024.

[6]. Chen, Yinghao, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. "Chatunitest: A Framework for LLM-Based Test Generation." In Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, 572-576. 2024.

[7]. Pizzorno, Juan Altmayer, and Emery D. Berger. "Coverup: Coverage-Guided LLM-Based Test Generation." arXiv preprint arXiv:2403.16218 5 (2024).

[8]. Gu, Siqi, Quanjun Zhang, Kecheng Li, Chunrong Fang, Fangyuan Tian, Liuchuan Zhu, Jianyi Zhou, and Zhenyu Chen. "Testart: Improving LLM-Based Unit Testing

via Co-Evolution of Automated Generation and Repair Iteration." arXiv preprint arXiv:2408.03095 (2024).

[9]. Ryan, Gabriel, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. "Code-Aware Prompting: A Study of Coverage-Guided Test Generation in Regression Setting using LLM." Proceedings of the ACM on Software Engineering 1, no. FSE (2024): 951-971.

[10]. Storhaug, André, and Jingyue Li. "Parameter-Efficient Fine-Tuning of Large Language Models for Unit Test Generation: An Empirical Study." arXiv preprint arXiv:2411.02462 (2024).

[11]. Pan, Rangeet, Myeongsoo Kim, Rahul Krishna, Raju Pavuluri, and Saurabh Sinha. "Multi-Language Unit Test Generation using LLMs." arXiv e-prints (2024): arXiv-2409.

[12]. Zhang, Yuwei, Qingyuan Lu, Kai Liu, Wensheng Dou, Jiaxin Zhu, Li Qian, Chunxi Zhang, Zheng Lin, and Jun Wei. "Citywalk: Enhancing LLm-Based C++ Unit Test Generation via Project-Dependency Awareness and Language-Specific Knowledge." ACM Transactions on Software Engineering and Methodology (2025).

[13]. Bhatia, Shreya, Tarushi Gandhi, Dhruv Kumar, and Pankaj Jalote. "Unit Test Generation using Generative AI: A Comparative Performance Analysis of Auto Generation Tools." In Proceedings of the 1st International Workshop on Large Language Models for Code, 54-61. 2024.

[14]. Zhong, Zhiyuan, Sinan Wang, Hailong Wang, Shaojin Wen, Hao Guan, Yida Tao, and Yepang Liu. "Advancing Bug Detection in Fastjson2 with Large Language Models Driven Unit Test Generation." arXiv preprint arXiv:2410.09414 (2024).

[15]. Bayrı, Vahit, and Ece Demirel. "AI-Powered Software Testing: The Impact of Large Language Models on Testing Methodologies." In 2023 4th International Informatics and Software Engineering Conference (IISEC), 1-4. IEEE, 2023.