PyTestGuard: An IDE-Integrated Tool for Supporting Developers with LLM-Generated Unit Tests

Master's Thesis

Nada Mouman

PyTestGuard: An IDE-Integrated Tool for Supporting Developers with LLM-Generated Unit Tests

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Nada Mouman
born in Pavia, Italy

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

PyTestGuard: An IDE-Integrated Tool for Supporting Developers with LLM-Generated Unit Tests

Author: Nada Mouman
Student id: 4996305

Abstract

Unit testing is an important step in the software development workflow to detect bugs and ensure system correctness. Recently, Large Language Models (LLMs) have been explored to automate unit test generation and have demonstrated promising results. However, the generated tests are not always reliable, as they may contain syntax errors, hallucinations, test smells, or failing assertions. We conjecture that providing developers with feedback on such issues will increase the adoption of LLMs in real-world workflows. To address this, we propose PyTestGuard, a PyCharm plugin that allows developers to generate and refine unit tests directly within the Integrated Development Environment (IDE). Beyond test generation, PyTestGuard helps users evaluate test quality by detecting test smells and reporting issues such as missing arguments or references to non-existing objects. We conducted a user study with nine participants to assess PyTestGuard's usefulness as a testing assistant and to identify areas for improvement. Participants reported that the tool's feedback on test quality, along with its summarised error messages and coverage information, supported them while writing unit tests. However, they also faced challenges and suggested improvements before completely trusting LLM-based test generation in their development workflow. Based on these findings, we highlight several design recommendations for future tools that aim to integrate LLMs into software testing workflows.

Thesis Committee:

Chair:	Dr. A. Panichella, Faculty EEMCS, TU Delft
University supervisor:	Dr. C. Brandt, Faculty EEMCS, TU Delft
Committee Member:	Prof. Dr. W.P. Brinkman, Faculty EEMCS, TU Delft

Preface

Writing this preface marks one of the last steps in finishing my master's thesis, and also in wrapping up my time as a student. Just the other day, while walking around campus, I was thinking about how quickly these six years have gone by. Seeing the new first-year students made me remember when I was in their shoes, trying to find my way in a completely new place. Working on this thesis over the past seven months has been a lot of fun. I have always been interested in software testing, and getting the chance to combine it with large language models made the whole process even more enjoyable.

First, I would like to thank my supervisor, Carolin Brandt, for proposing the initial draft of this project and helping me shape it into its current form. I am also very thankful for her guidance and feedback, from the stages of designing PyTestGuard to the process of writing my thesis. Our weekly meetings were always helpful, and I appreciate how she was always available to answer my questions and discuss my ideas. I also want to thank my thesis advisor, Annibale Panichella, for his valuable feedback and suggestions, which helped me to strengthen and improve my thesis.

Finally, I want to thank my family for always supporting me throughout this journey. My dad, for always being there whenever I needed help, and my mom for teaching me the importance of education. I would like to thank my sister for taking the time to proofread this thesis and suggesting fixes that made it more readable. I also want to thank my brother for giving me feedback while I was designing PyTestGuard and helping me choose the title of this thesis.

Nada Mouman
Delft, the Netherlands
September 4, 2025

Contents

List of Figures

Chapter 1

Introduction

Unit testing is one of the most common types of tests in software development. According to the Developer Ecosystem Report 2024 [1], 78% of tests written by developers are unit tests. These tests verify the behaviour of individual components, which can help detect early bugs in production code and reduce the cost of fixing them later [33]. However, developers often choose to skip or postpone writing unit tests as they can be time-consuming [25].

In recent years, there has been a growing trend toward using Large Language Models (LLMs) to automate common and repetitive tasks in software development. Research indicates that LLMs can help in tasks such as code summarisation [3, 49], code generation [44], and unit testing [5, 63, 93]. By automating test generation, LLMs could save developers time and effort during the development process. However, recent studies show that instead of reducing effort, developers often spend a significant amount of time reviewing and fixing LLM-generated output [15, 23, 88]. This shift from creating tests to reviewing them [83] introduces new challenges that existing tools do not sufficiently support.

LLM-generated Python tests often suffer from quality issues such as test smells. Test smells result from developers' suboptimal design decisions in test code, and they can reduce test readability and maintainability. Notably, recent studies have found that test smells are common in LLM-generated tests. In an empirical study, Alves et al. [7] discovered that nearly half (47.4%) of the test cases generated by GitHub Copilot contained at least one test smell. Ziberman et al. [108] also found that GPT-generated test suites had an average of 5.6 test smells for GPT-3, 9.4 for GPT-3.5, and 8.3 for GPT-4. Although developers might perceive test smells as having "low severity" [21], they can still negatively impact the maintainability of the test code [13]. In some cases, they might fail to detect defects in production code [84], especially when important assertions are missing in the test cases [38].

In addition to test smells, LLM-generated Python tests are also inclined to have other quality issues. Recent studies [16] have found that the majority of LLM-generated test cases contained syntax errors, incomplete assertion statements, and missing imports. El Haji et al. [32] also report that 54.72% of Python tests generated by GitHub Copilot in an existing test suite failed or were broken. In contrast, for those generated without an existing test, only 7.55% passed. One of the main reasons the tests were not correct was the presence of hallucinations. These can include non-existent objects, incorrect types used in a function call, and non-existent or incorrect attributes for an object [54, 106].

While these quality issues are common, current LLM-based tools provide little assistance in identifying and correcting them. Despite attempts having been made to introduce metrics for evaluating the quality of LLM-generated tests [2, 56, 67], these metrics have not yet been incorporated into existing LLM-based tools in a way that provides real-time feedback on errors, test smells, or hallucinations. Without proper support, developers' trust in LLM-generated tests may decrease because of these issues [11, 30], and the manual effort required for debugging and fixing the test cases may increase, as these shortcomings can lead to compilation or runtime errors. Combined with test smells, the process of reviewing and correcting LLM-generated tests becomes difficult and ineffective, showing a need to integrate feedback directly into developers' Integrated Development Environments (IDEs), making it easier to locate errors, detect test smells, and fix hallucinations in real-time.

To address these challenges, our vision is to help developers use LLMs more effectively for unit test generation by providing real-time and actionable feedback directly within their IDE. By integrating the functionality to detect or highlight quality issues present in LLM-generated tests into their development environment, developers could more easily review them. This integration could also help increase their trust in using LLM-generated tests in their development workflow.

In this thesis, we aim to support developers by providing feedback on the quality of LLM-generated tests. To achieve this, we develop and evaluate a new PyCharm[1] plugin, PyTestGuard, which can automatically generate unit tests and highlight quality issues such as syntax errors, missing imports, test smells, or hallucinations. We focus on Python because it is one of the most popular programming languages [1, 64], which makes our work relevant for real-world development.

1.1 Research Questions

To achieve our vision of helping developers use LLMs for more effective unit test generation, we must first investigate how they interact with such tests in IDE tools that provide quality feedback. In this thesis, we examine the kinds of support developers expect from such tools, focusing on the challenges they face. We also investigate whether a tool like PyTestGuard can increase their willingness to adopt LLM-generated unit tests. Based on this, we propose the following research question:

Research Question 1

What support do developers expect in an IDE tool like PyTestGuard to assist them in using LLM-generated unit tests?

We aim to investigate what type of support, such as coverage and quality feedback, developers expect from an IDE tool, similar to PyTestGuard, when generating unit tests using large language models. We address RQ1 by evaluating whether our tool meets their expectations and whether it can be enhanced with additional features to meet their needs.

[1] https://www.jetbrains.com/pycharm/

Research Question 2

What challenges do developers face when using the simple version of PyTestGuard compared to the enhanced version?

With RQ2, we want to explore the challenges developers encounter when interacting with LLM-generated unit tests. These challenges could include missing edge cases, incorrect logic, or syntax errors. We compare two versions of PyTestGuard: a simple one that only generates unit tests, and an enhanced version. The enhanced version also provides quality information such as test smells, statement coverage, and the presence of hallucinations. By analysing these challenges in both versions, we can evaluate whether the enhanced version provides sufficient support for developers to understand and work with the generated tests. Additionally, we could also discover other needs that are not met by our tool, which would offer valuable insights for improving future tools.

Research Question 3

How does PyTestGuard affect developers' willingness to use LLM-generated unit tests?

RQ3 evaluates the tool's impact on developers' perceptions of LLMs. One of our goals is to encourage developers to use LLMs by leveraging the advanced features of PyTestGuard. Therefore, we aim to understand whether the tool increases trust in LLM-generated tests and motivates developers to include them in their development workflow.

1.2 Research Approach

To identify the key features required for a developer-centric tool to generate unit tests in Python, we first designed and implemented PyTestGuard. It is a plugin for PyCharm that allows users to generate test cases for specific Python methods. The tool also supports test smell detection, which can be used to assess the quality of the generated tests. Users can also execute the tests and view both statement coverage and the change in statement coverage, which indicates the incremental percentage added to the original test suite. Visual feedback about the execution of the tests is displayed, along with a summarised version of the execution message when a test fails. Developers can also directly edit the generated test cases within the tool. Additionally, the plugin highlights common shortcomings often found in LLM-generated unit tests.

To answer our research questions, we conducted a user study where we invited developers to use two versions of PyTestGuard to generate test cases. These versions include a simple version that only generates unit tests and an enhanced version that, in addition to test generation, provides quality feedback. We conducted a semi-structured interview, where participants not only tried out PyTestGuard but also completed a questionnaire. The questionnaire consisted of three main parts. First, we used a pretest questionnaire to ask participants about their experience with testing and their familiarity with using large language models for unit testing. The second part consisted of completing tasks using both a simple and enhanced version of PyTestGuard to generate test cases. After each task, we asked participants to provide feedback on their experience. Finally, we concluded with a posttest questionnaire to collect general feedback on the tool's features and overall user experience.

We addressed RQ3 by asking the participants to rate their likelihood of using LLMs for unit testing before and after using PyTestGuard. We, then, compared these ratings to evaluate changes in their willingness to adopt LLMs. To answer RQ1 and RQ2, we analysed the responses from the entire questionnaire.

We provide the following contributions in this thesis:

1. PyTestGuard: a plugin for PyCharm that generates unit tests for Python methods. It also provides test smell detection and highlights for general shortcomings found in LLM-generated unit tests.

2. Insights into how adding test smell detection and inspections, specifically designed for LLM-based tests, influences developers' adoption of LLM-generated tests in their workflows.

3. Suggestions for additional features needed to create an effective LLM-based unit test generation tool for developers, providing quality feedback that includes test smell detection and highlighting common issues such as hallucinations and wrong types.

1.3 Thesis Overview

The rest of the thesis is structured as follows. Chapter 2 presents the background and related work on unit test generation and test smells. In Chapter 3, we describe the design and implementation of PyTestGuard. Chapter 4 describes the user study we conducted to evaluate our tool. In Chapter 5, we present the results of the user study, while in Chapter 6, we analyse and discuss them to answer RQ1-RQ3. Finally, we conclude the thesis in Chapter 7 and discuss some prospects for future work.

Chapter 2

Background and Related Work

This chapter presents existing research on automated test generation, starting with traditional approaches. We then examine recent work that leverages large language models for generating unit tests, followed by IDE-integrated approaches that aim to support automated test generation within developers' workflows. Finally, we present background information about test smells and prompt engineering, and how they relate to our thesis.

2.1 Automated Test Generation Tools

Besides using large language models to generate unit tests, several other approaches have been proposed to improve the coverage of software under test. These techniques typically generate unit tests using search-based [28, 36] or random-based methods [65]. In this section, we mainly focus on approaches for Python, as our tool, PyTestGuard, is specifically designed for Python projects.

One of these tools is Pynguin [57], which is a test generation framework for Python. It uses search-based test generation techniques to generate tests randomly and iteratively refines them to maximise code coverage. A Python module is first taken as input, and then Pynguin analyses it to extract the relevant information before generating the corresponding test suite. The information includes the methods and types of the module, as well as its dependencies. Although Pynguin is capable of achieving high coverage, one of its limitations is that it can only be run as a command-line application. Alternatively, it can be accessed as a library in other projects via its public API.

Another approach to improving test suites is test amplification. Compared to test generation, which creates tests from scratch based on the code under test, amplification extends existing handwritten tests to cover additional corner cases [26]. AmPyfier [80] introduces test amplification to Python by using information available during runtime. It was evaluated on seven open-source projects, and the results showed that it successfully strengthened 7 out of 10 test classes. The mutation score improved from 0.74% in well-tested projects to 53.06% in poorly tested ones.

Despite traditional approaches yielding good results in terms of coverage, studies have highlighted their limitations in terms of readability and comprehensibility [37, 66]. Grano et al. [40] discovered that the generated tests by EvoSuite [36], the state-of-the-art search-based test generation tool for Java, are typically less readable than those written by humans. Bathia et al. [17] compared ChatGPT and Pynguin in terms of their ability to generate unit tests for Python. They discovered that both tools had nearly identical coverage for both small and large samples, and approximately one-third of the assertions generated by ChatGPT for certain categories were incorrect. Tang et al. [86] also conducted a comparison between ChatGPT and EvoSuite for generating JUnit test cases. In general, EvoSuite performed better for large classes, achieving an overall statement coverage of 74.2%, compared to 55.4% for ChatGPT. Although ChatGPT had lower test coverage, EvoSuite suffered from readability issues, and execution times and computation expenses increased with code complexity.

There have been attempts to combine search-based software testing with large language models to enhance the understandability of automatically generated test cases. For example, Deljouyi et al. [27] attempted to utilise LLMs to enhance the names of identifiers generated by EvoSuite and to add descriptive comments to the test code. Their approach, UTGen, generated 8,430 tests with a pass rate of 73.27% compared to EvoSuite's 8,315 tests at 79.01%. Despite this, UTGen tests helped participants to fix up to 33% more bugs and complete tasks up to 20% faster than with baseline test cases, while improving the clarity of the test cases. Similarly, Lemieux et al. [50] built CODAMOSA, an extension of Pynguin that leverages LLMs to generate test cases. When Pynguin stalls in its exploration, CODAMOSA uses OpenAI's Codex [22] to generate additional test cases for under-covered areas. In an evaluation over 486 benchmarks, CODAMOSA improved coverage on 173 benchmarks compared to SBST and 279 benchmarks compared to LLM-only tests, while reducing coverage on only 10 and 4 benchmarks, respectively.

Although tools like UTGen and CODAMOSA use LLMs to enhance test generation, they mainly focus on improving readability or generating additional tests when search-based approaches stall. PyTestGuard differs by providing developers with feedback on test quality, such as the presence of test smells and hallucinations, in addition to generating unit tests with LLMs.

2.2 Large Language Models for Unit Test Generation

Recent work has increasingly explored the use of large language models (LLMs) to automatically generate unit tests, investigating their capabilities, limitations, and techniques to improve the quality of the generated tests.

Schäfer et al. [79] introduced TestPilot, an iterative LLM-based test generation tool for JavaScript that generates unit tests for methods. Unit tests are generated by iteratively querying an LLM with a prompt containing signatures of the functions under test, and optionally, the prompt can also include the bodies and documentation related to these functions. In the event of failure, TestPilot creates a new prompt with the test and the error message and asks the model to fix the failed test. They achieved a median statement coverage of 70.2% and a branch coverage of 52.8% on 25 npm packages with a total of 1,684 API functions. Other studies have also adopted iterative prompting strategies, but instead of providing error messages, they provide coverage information from the generated test cases as feedback to enhance the quality of the test cases. For instance, TestART [41] incorporates such feedback into its iterative loop for generating JUnit tests, achieving a 78.55% pass rate across three open-source and industrial datasets, which is 18% higher than the pass rate achieved with GPT-4.

Following this approach, CoverUp [5] also incorporates coverage analysis into its prompts to iteratively generate Python regression tests. Unlike other tools that rely on method signatures, CoverUp targets code segments lacking coverage and explicitly requests tests to address those gaps. To handle missing dependencies, it utilises static analysis to generate the necessary import statements. In case of build failures, failing tests, or insufficient coverage, it continues the conversation with the LLM to refine the output. Tests that fail are excluded from the final results, even though some could potentially add value to the suite. While these tools demonstrate success in generating high-coverage test cases, they lack integration with existing IDEs or focus exclusively on JavaScript and Java, as seen with TestPilot and TestART, respectively.

Several empirical studies have evaluated LLMs on how well they perform on generating unit tests, focusing on correctness, coverage, and test smells present in the tests. For instance, Yang et al. [101] conducted an empirical study with five open-source LLMs, including CodeLlama [75], to assess their ability to generate unit tests for Java projects. They found that, compared to traditional test generation tools such as EvoSuite, LLM-generated tests often fail to compile due to hallucinations, resulting in errors like unresolved symbols, parameter mismatches, and abstract instantiations. Therefore, traditional tools achieve higher test coverage under these conditions. Correspondingly, Siddiq et al. [82] assessed the test generation abilities of Codex [22], GPT-3.5-Turbo, and StarCoder [51] on Java classes. They found that one of the root causes of compilation errors in tests is the use of unknown symbols, which are symbols not found in the code base. They accounted for more than 62% of compilation errors. Generated tests also showed test smells, with Assertion Roulette and Magic Number Test being the most common, ranging from 15% to 61.3% and 21.8% to 100% in frequency, respectively, across the models. Similarly, Yuan et al. [103] conducted a quantitative analysis and a user study to investigate the quality of tests generated by ChatGPT in terms of correctness, sufficiency, readability, and usability. They found that the generated tests suffer from compilation errors and execution failures. They compared the ChatGPT-generated test to existing learning-based and search-based techniques, AthenaTest [87] and EvoSuite, and they observed that executable tests generated by ChatGPT achieved the highest code coverage. They also found that ChatGPT tests were most comparable to manually written tests in terms of the number of assertions per test, whereas AthenaTest generated more assertions and EvoSuite generated fewer assertions.

While most of the empirical evaluations discussed so far are conducted on Java unit tests, El Haji et al. [32] investigated the ability of GitHub Copilot to generate unit tests for Python. They investigated the usability of 290 generated tests and found that only 7.55% of the tests generated without an existing test suite were passing, while those with an existing test suite had a passing rate of 45.28%. Most of the tests were failing, broken or empty. Taken together, these findings suggest that current tools for automated LLM-based test generation should incorporate feedback mechanisms regarding test quality, such as compilation success, execution results, coverage metrics and presence of test smells. These quality indicators could improve the value of the generated tests to the developers. Our tool, PyTestGuard, addresses this gap by not only generating unit tests but also providing quality feedback on the presence of test smells and hallucinations, such as non-existing objects and missing arguments.

Some tools have attempted to provide quality feedback for LLM-generated tests. For example, AgoneTest [56] is an automated tool that generates and evaluates test suites for Java projects at the class level. It helps developers assess LLM-generated test quality by analysing coverage metrics such as line, method, branch, instruction, and mutation coverage. AgoneTest also detects common test smells, such as Assertion Roulette, Default Test, Duplicate Test, and Eager Test, which can affect test quality. To achieve this, AgoneTest uses tools such as JaCoCo for measuring code coverage, PiTest for mutation testing, and TSDetect [70] for identifying test smells. It automatically collects information from their reports and compiles the data for each class in the project. After generating the tests, AgoneTest creates a report and a CSV file summarising coverage metrics and test smells for each LLM and prompting technique, helping developers understand and improve the generated tests. Although the tool provides developers with feedback about the quality of the generated tests, it is not yet integrated within any IDE and supports only Java classes. We build on this idea by introducing PyTestGuard, a similar approach for Python that not only provides LLM-based test generation and quality feedback but is also integrated directly into the IDE to support developers within their development workflow.

2.3 IDE-Integrated Test Generation Tools

Several tools have explored IDE-integrated approaches for automated test generation. For example, Arcuri et al. [9] have developed plugins to integrate EvoSuite with Maven, IntelliJ IDEA, and Jenkins to allow the automation of test generation for JUnit tests within industrial projects. These plugins allow developers to run EvoSuite as part of the build process, monitor test results in the IDE, and integrate generated tests into the source tree.

Similarly, TestSpark [78] is an IntelliJ plugin that enables users to generate unit tests directly within the IDE. It also allows users to easily modify and run each generated test and integrate them into the project workflow. TestSpark supports both test generation by EvoSuite [36], the state-of-the-art search-based test generation tool for Java, and large language models. Its LLM-based generation process is iterative. When the LLM returns the generated tests, they are executed, and if they fail, a new prompt is constructed containing the compilation error. This prompt is then sent back to the LLM for correction. If, after a maximum number of iterations with the LLM, the tests still fail to compile, TestSpark returns them to the users along with an error message.

Another example is TestCube [19], an IntelliJ plugin built on top of DSpot that focuses on developer-centric test amplification by generating amplified test cases that are easier to understand. TestCube also provides an integrated exploration environment within the IDE to help developers adopt the amplified tests into their existing test suites. Although these tools improve IDE integration, they currently support only Java or Kotlin and do not provide quality feedback on the generated tests. In contrast, PyTestGuard supports Python and provides detailed quality feedback, including test smell detection and the identification of hallucinations.

Zhao et al. developed CodingGenie [107], an LLM-powered programming assistant for VS Code that automatically offers chat suggestions by analysing the provided context. These suggestions include code improvements, code explanations, brainstorming ideas, and unit testing. The tool uses state-of-the-art LLMs, such as GPT-4 and Claude Sonnet-3.5, prompting them with 500 characters of code above and below the user's cursor position. The user's message history is also included to inform the LLM of the types of suggestions that might be relevant. While CodingGenie provides tailored suggestions within the IDE, its generated tests may still suffer from quality issues such as hallucinations and test smells, which PyTestGuard can directly inform developers about.

2.4 Test Smells

The concept of test smells was first introduced by van Deursen et al. [89], who defined them as poor design choices or patterns in test code. They created a catalogue of 11 test smells and suggested refactoring operations to remove them. This catalogue was later expanded by Meszaros [60] to include 18 test smells. Such test smells include Assertion Roulette, where multiple assertions have no explanation, and the Magic Number Test, where a test case contains undocumented numerical values. Test smells can also negatively impact the readability and maintainability of tests [13, 69], as well as their reliability, since they can affect the detection of defects [84].

In 2012, Bavota et al. [12] introduced one of the first test smell detection tools. It uses hand-crafted static detection rules to identify nine types of test smells in Java test suites. They reported achieving 100% recall and 88% precision when detecting test smells in test cases written by developers. Virginio et al. [91] later presented JNose Test, a test smell detection tool capable of detecting 21 distinct types of test smells for Java projects. It can also present the number of test smells detected per class and several code metrics. These include cyclomatic complexity, branch, instruction, and method coverage, using the JaCoCo library. They analysed the influence of test smells on code coverage and concluded that these smells might affect test coverage. Around the same time, Peruma et al. [70] released tsDetect, which can detect 19 types of test smells. This tool uses an abstract syntax tree to analyse JUnit test suites and reports an average F-score of 96.5% for each smell type. However, these tools are limited to Java and are not integrated within any IDE. Santana et al. addressed this issue by introducing RAIDE [77], a plugin for Eclipse, but it is currently limited to only two types of test smells: Assertion Roulette and Duplicate Assert.

Most test smell detection tools are designed to detect test smells in Java test suites, but there is limited support for other popular languages, such as JavaScript and Python [4]. There are currently three test smell detection tools for Python: PyNose [94], TEMPY [35], and Pytest-smell [35]. Pytest-smell is a tool that automatically detects test smells written with pytest. It is a Command-Line Interface (CLI) library that supports the detection of 10 different smell types. TEMPY, on the other hand, is not integrated into any IDE. Instead, it is run separately and generates a standalone report detailing the test smells found in the file. TEMPY supports both pytest and unittest libraries and can also detect 10 different types of test smells. Finally, PyNose is a plugin for PyCharm, which directly detects and embeds test smells into the codebase by highlighting where they occur in the main editor. It can support the detection of 17 test smells.

Some studies [7, 82, 108] have shown that LLM-generated tests can also contain test smells, such as Assertion Roulette and Magic Number Test. To address this, PyTestGuard provides test smell detection as part of its quality feedback, which helps developers identify and correct these issues directly within the IDE while generating unit tests.

2.5 Prompt Engineering

With the advancement of LLMs, researchers have started investigating various prompting techniques for instructing the models rather than pre-training or fine-tuning them for specific tasks [55, 81]. Prompt engineering is an important technique for working with pre-trained LLMs, such as GPT-4. Input queries, called ”prompts,” guide the model's behaviour by setting rules or tailoring outputs. This is done without changing the model itself. Effective prompts can improve the accuracy and usefulness of the model's responses [58].

Recently, prompt engineering has proven effective in generating unit tests [93]. For example, Siddid et al. [82] used the zero-shot technique to instruct the LLM. The prompt includes the full code of the class under test, package declarations, and the necessary imports for a compilable test file. It also includes inline comments about the name of the test class to be generated (e.g., `${className}${suffix}Test.java`). Zero-shot learning [48] provides the model with only a task description and requests a response from the LLMs. Rather than relying on labeled input-output pairs, the model uses its internal knowledge to generate effective responses.

Few-shot learning [20], on the other hand, provides a set of high-quality examples, each consisting of both the input and the corresponding desired output, to the model. These examples help the model understand what is needed and can lead to better results, though this approach uses more input tokens and may introduce bias based on the chosen examples [76].

Chain-of-Thought (CoT) [97] prompting is a technique where the model is asked to break down a problem into smaller steps and explain its thought process as it proceeds, rather than providing the answer immediately. This makes it easier to follow the logic, which is particularly useful for tasks such as math problems or those that require multiple steps to solve. Tree-of-Thought (ToT) prompting [102] extends this by allowing the model to explore different paths to solve a problem, much like exploring branches on a tree. The model examines each option and selects the best one, which is especially useful for complex or creative problems. Self-Consistency [95] is another prompting technique that helps make CoT responses more reliable. Instead of just picking the first answer the model comes up with, the model tries out different ways to solve the problem and then selects the answer that appears most frequently. This approach employs different methods to find the solution, which has been demonstrated to be more effective on problems with multiple correct solutions.

In this thesis, we focus on using Chain-of-Thought (CoT) prompting in PyTestGuard to generate unit tests. By guiding the LLM to reason step by step about the code under test, our tool may generate tests that are more accurate and correct [63].

Chapter 3

Design and Implementation of PyTestGuard

This chapter presents the design and implementation of PyTestGuard, a unit test generation tool for Python. We explain in detail the different tool features and their development. Key features include the detection of test smells, the identification of common issues in unit tests generated by large language models (LLMs), and the calculation of test coverage. We also describe the simple mode of PyTestGuard, which can be enabled to generate unit tests only, without providing quality feedback or allowing test execution.

3.1 Overview of PyTestGuard

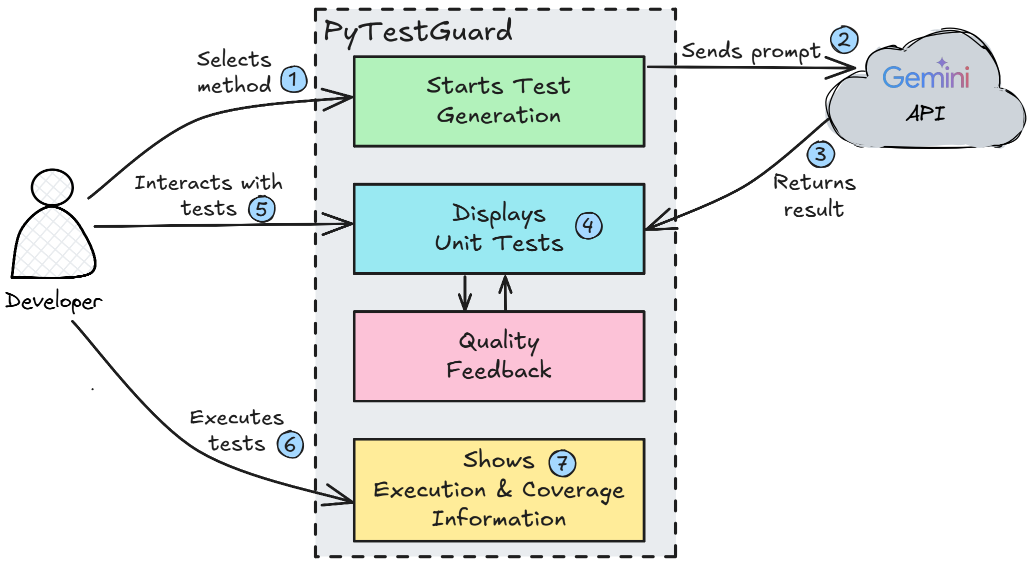
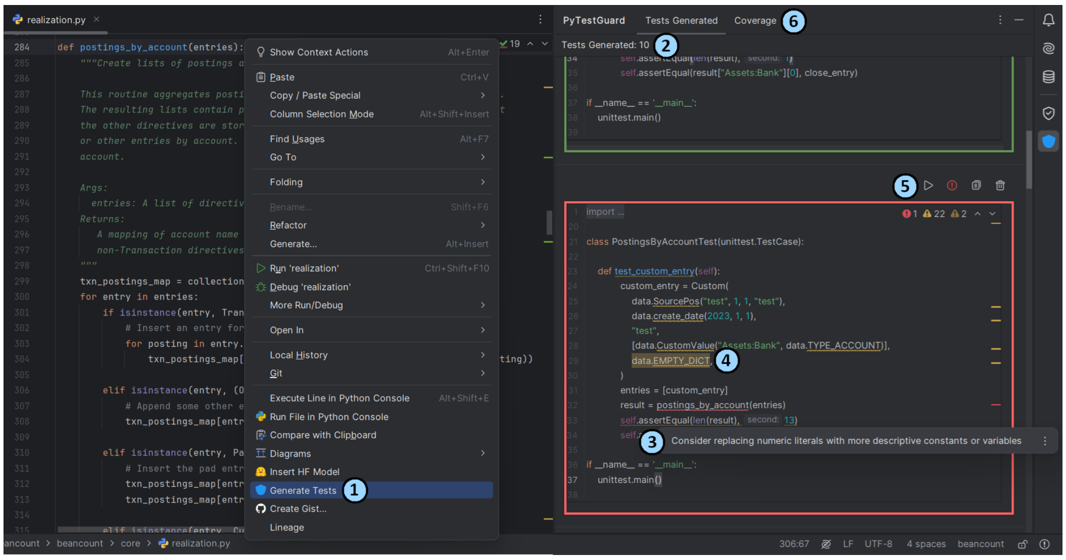


Figure 3.1: Overview of PyTestGuard workflow.

To help developers generate unit tests for Python functions using LLMs, we created a PyCharm plugin called PyTestGuard. A user can select a method to put under test and then run PyTestGuard to generate unit tests. PyTestGuard sends a prompt containing the method body to an LLM, which returns the generated tests. These tests are then parsed and displayed to the user directly within the plugin. Quality feedback about the generated tests, such as the presence of test smells, is also shown to the developer. The developer can then interact with the generated test cases from inside the tool. After editing a unit test, the user can run it with PyTestGuard, which provides both execution results and coverage information. Figure 3.1 shows an overview of the PyTestGuard workflow.



```
def postings_by_account(entries):
    """Create lists of postings a

    This routine aggregates posti
    The resulting lists contain p
    the other directives are stor
    or other entries by account.
    account.

    Args:
      entries: A list of directiv
    Returns:
      A mapping of account name
      non-Transaction directives
    """
    txn_postings_map = collection
    for entry in entries:
        if isinstance(entry, Tran
            # Insert an entry for
            for posting in entry.
                txn_postings_map[                    ting))

        elif isinstance(entry, (O
            # Append some other e
            txn_postings_map[entr

        elif isinstance(entry, Pa
            # Insert the pad entr
            txn_postings_map[entr
            txn_postings_map[entr

        elif isinstance(entry, Cu
```

```
        self.assertEqual(len(result),  second: 1)
        self.assertEqual(result["Assets:Bank"][0], close_entry)

if __name__ == '__main__':
    unittest.main()
```

```
import ...

class PostingsByAccountTest(unittest.TestCase):

    def test_custom_entry(self):
        custom_entry = Custom(
            data.SourcePos("test", 1, 1, "test"),
            data.create_date(2023, 1, 1),
            "test",
            [data.CustomValue("Assets:Bank", data.TYPE_ACCOUNT)],
            data.EMPTY_DICT,
        )
        entries = [custom_entry]
        result = postings_by_account(entries)
        self.assertEqual(len(result),  second: 13)
        self.

if __name__ == '__main__':
    unittest.main()
```

Consider replacing numeric literals with more descriptive constants or variables

Figure 3.2: Overview of PyTestGuard layout.

When a user wants to generate unit tests for a Python function, they can right-click on any line inside the function body and then click on *"Generate Tests"* (1 in Figure 3.2). A pop-up window will appear, as shown in Figure 3.3, prompting them to select the test framework for the unit tests. Users can choose between `pytest`[1] and `unittest`[2], and then confirm their selection by clicking the *"Ok"* button.

After that, the PyTestGuard tool window[3] appears on the right side of PyCharm (2 in Figure 3.2). The window displays all the unit tests generated by the LLM. At the top of the window, the number of generated tests is shown. Each test is shown separately in its own block. Above each test block, users have options to view, copy and remove the test. The test blocks also serve as independent editors, enabling developers to directly edit test cases within the tool without needing to copy them into the main editor. Users can also run the generated tests directly from the tool by clicking the *"Run Test"* button (5 in Figure 3.2). After the test execution, the border of the test will turn either green or red based on

[1]https://pytest.org/
[2]https://docs.python.org/3/library/unittest.html
[3]https://www.jetbrains.com/help/idea/tool-windows.html

the execution results. PyTestGuard also shows the coverage information about the executed test in the *Coverage* tab ((6) in Figure 3.2). For the plugin layout, we took inspiration from TestSpark [78], an IntelliJ plugin that generates unit tests for Java.

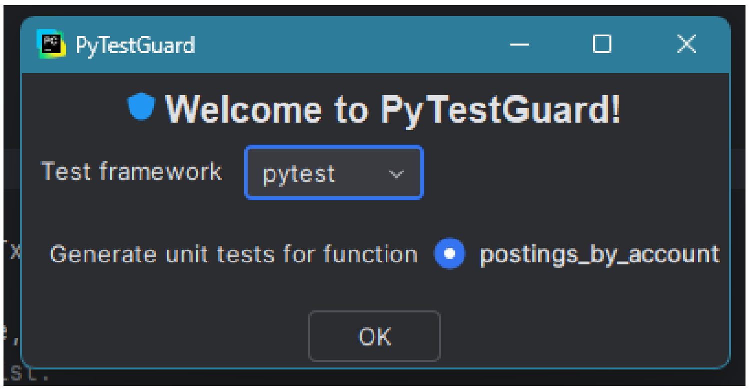


Figure 3.3: Popup window to confirm test generation.

PyTestGuard also includes test smell detection, which automatically highlights test smells present in the generated tests ((3) in Figure 3.2). It also supports automatic flagging of common shortcomings generally found in LLM-generated tests ((4) in Figure 3.2). These issues include syntax errors, incorrect input types, and non-existent objects. Users can directly inspect and fix the issues and the test smells within the plugin, providing them with more control over the quality and reliability of the generated tests.

Architecture and Components

The architecture of PyTestGuard follows an automated execution and visualisation workflow that integrates LLM-based test generation with test smell detection, which is shown in Figure 3.4.

The workflow starts in the `PromptGeneration` component when a method is selected for testing. This module constructs a prompt using information extracted from the method body and instructs the model to generate corresponding unit tests. The prompt is forwarded to the `GeminiRequestManager`, which acts as the intermediary between PyTestGuard and the external `Gemini API`. The manager formats and transmits the request to the API, which generates the required test cases.

Once Gemini's response is received, the `GeminiRequestManager` forwards it to the `TestAssembler`. The `TestAssembler` then parses and processes the generated tests into an executable form before sending them to the `DisplayManager` for user presentation. When a test case is executed, the `DisplayManager` oversees its execution through the `TestExecutor`, which runs the tests and returns execution results along with the statement coverage. The `DisplayManager` then displays these results to the user.

The `DisplayManager` also manages the `TestVisualizer`, which presents the generated tests in separate and independent boxes. It also allows test cases to be editable and is responsible for visualising the highlights of common issues found in LLM-based unit tests, such as hallucinations. Additionally, the `TestVisualizer` integrates PyNose, a Python test smell detection tool, to visualise the presence of test smells in the generated test cases.

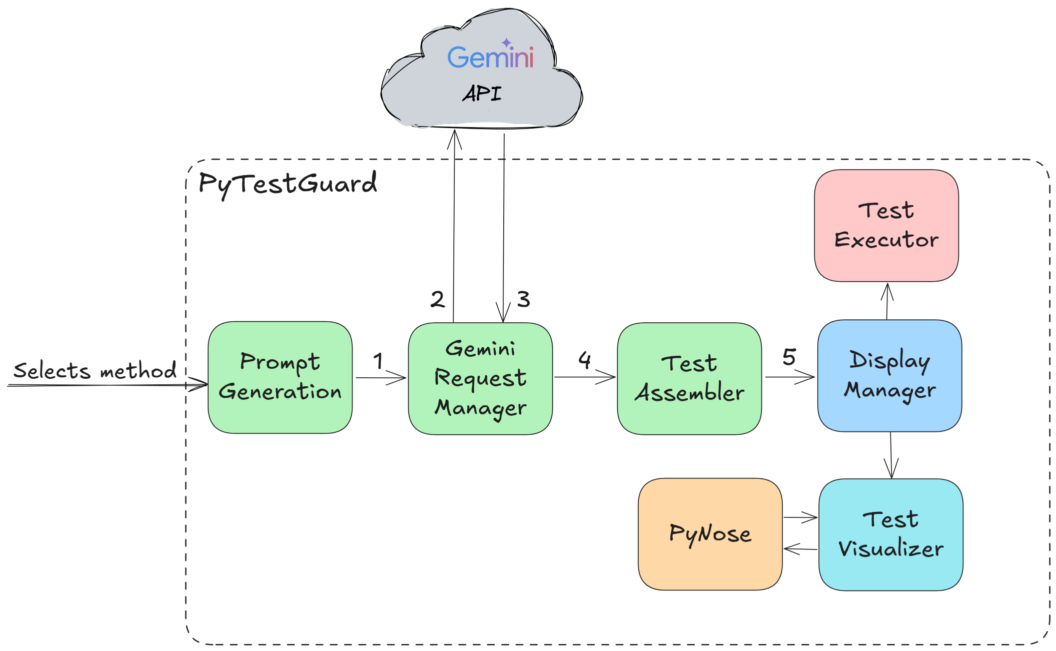


Figure 3.4: Overview of core components of PyTestGuard.

3.2 Generating Unit Tests

Once the user confirms test generation, PyTestGuard constructs a user prompt and retrieves the system prompt to create the request body that will be sent to the LLM. We selected Gemini-2-Flash[4] as the model for generating unit tests. This model supports a 1M token context window, which is ideal when the method under test is large. We send the prompts to the model by using the Gemini Developer API, with a temperature setting of 0.5 [41]. This achieves a balance between allowing some creativity in the model's output while maintaining reasonably deterministic responses [92]. We, then, parse the response from the API to extract the unit tests, which are passed to the `PyTestGuardDisplayManager`. This class is responsible for rendering the tool window and displaying the generated tests.

Prompt creation

The prompts used to generate Python unit tests with Gemini are shown in Figures 3.5 and 3.6. We designed them to guide the model through a structured process, resulting in accurate and complete unit tests.

[4]https://cloud.google.com/vertex-ai/generative-ai/docs/models/gemini/2-0-flash

System Prompt

You are a Python developer with a great experience on writing unit tests. Your job is to only provide unit tests for Python code.

Figure 3.5: System prompt used for generating unit tests by the LLM.

We use the persona pattern for the system prompt [5, 92] by defining the model as an experienced Python developer specialising in unit testing. This enables the model to produce relevant, professional test code without any extra explanations.

We start the user prompt by instructing the model to generate unit tests for the specified Python method using the selected testing framework, following the instructions provided in the rest of the prompt (① in Figure 3.6). We use chain-of-thought (CoT) reasoning for the user prompt by providing step-by-step instructions that guide the model to analyse the method [5, 27, 92]. We instruct the model to identify edge cases, verify imports, follow method signatures, and follow the conventions of a specified testing framework ②. This guided description can help the LLM to yield better accuracy and correctness for the generated tests with respect to other prompting techniques [63].

To improve the model's context awareness, we have included the signatures of other methods present in the same file as the method under test in the prompt [103]. This information helps the model understand dependencies, parameter types, and expected behaviours, leading to more trustworthy unit tests. We use a prompt template to provide the required details [82], such as imported modules and global fields ③. It also instructs the model to output only Python code enclosed in backticks without any explanations ④, guaranteeing the output is clean and ready to use. At the end of the prompt, we provide the complete code of the method under test to the model ⑤.

These parts create a complete and clear prompt that guides the model to generate good unit tests aligned with the current coding standards and practices.

Implementation. We construct the prompt sent to the LLM by extracting the necessary context from the method code. We use the class `PsiHelper`, which first checks if there's a function where the user's cursor is currently located. If there is a function at that position, a `PsiFunctionContext` object is created. This object contains the method's name, its body, the names of its parameters, and, if available, the types of its parameters and the return type.

The `PsiFunctionContext` also extracts the signatures of other methods and the import statements from the same Python file. This information is stored in a `PsiFileContext` instance inside the `containingFile` attribute. If the method belongs to a class, we also gather information about that class, like its name and the names of its attributes. This data is stored in a `PsiClassContext` instance in the `containingClass` attribute.

Finally, the `PsiFunctionContext` object is passed to the `PromptGenerator`, which uses it to fill out the template used to build the user prompt for the model.

User Prompt

1

Create unit tests for the Python method `$METHOD_NAME` provided based on the given information using `$TESTING_PLATFORM`.

2

To achieve this, you must follow the following steps:
1) Analyze the provided method carefully to understand its functionality
2) Identify edge cases, such as boundary values and failure scenarios, and include them in the test suite
3) Check the provided imports and ensure they are correctly used in the unit tests
4) Ensure the method signatures are correctly followed
5) Use only `$TESTING_PLATFORM` for writing unit tests and adhere to its conventions
6) Follow the best coding practices, such as using meaningful test names and assertions
7) Formulate the test cases and make sure that each test is correct

3

You can use the following information for the unit tests:
`$IMPORTED_MODULES`
`$GLOBAL_FIELDS`
`$METHOD_SIGNATURES`

4

Answer ONLY with Python code, enclosed in backticks, without any explanation.

5

You can find below the method to be tested:

```python
def $METHOD_NAME:
        # Code under test
```

Figure 3.6: User prompt used for generating unit tests by the LLM.

3.3 Post-Generation Validation

PyTestGuard provides post-generation validation for the generated unit tests. It can provide inspections about the test quality, such as test smells. It also highlights issues usually found in LLM-based unit tests.

3.3.1 Common shortcomings in LLM-generated tests

To help developers identify quality issues in LLM-generated unit tests, PyTestGuard provides inspections that highlight problematic habits often found in generated test code. We identified these shortcomings through a review of existing literature on LLM-based test generation, using keywords such as *"bugs"*, *"hallucinations"*, *"quality assessment"*, *"automatic test generation"*, *"large language models"*, *"empirical evaluation"*, and *"code generation"*. Based on findings from prior work, we compiled a list of common issues that our tool identifies through visual warnings, as shown in Figure 3.7.

Syntax Errors. These include errors such as missing parentheses or colons, or indentation errors that prevent the code from running [32, 85, 105]. PyTestGuard can detect and highlight these errors to prevent execution failures.

Wrong Input Type. LLMs might generate functions with input types that do not match the expected types of method parameters [85]. For example, passing a string instead of an integer. If the type of the parameters is available in the function signature, our tool can find these mismatches and highlight them in the test. However, in the absence of type annotations, these errors generally occur at runtime as TypeErrors, which are then displayed in the summarised error message, as mentioned in Section 3.4.

Empty or Incomplete Generation. Some generated tests are empty or include only partial code, such as a test method without a body, or missing assertions [32, 85]. Our tool filters out any empty generations before displaying them to the users.

Incorrect Parameters. Unit tests might call methods with the wrong number of arguments. For example, using extra arguments and omitting required ones [32, 54, 85, 106]. PyTestGuard highlights these types of issues directly on the unit tests.

Hallucinated Objects. LLMs can sometimes hallucinate variables, classes, or functions that are not defined in the codebase [32, 85]. PyTestGuard highlights unresolved references as warnings for the user.

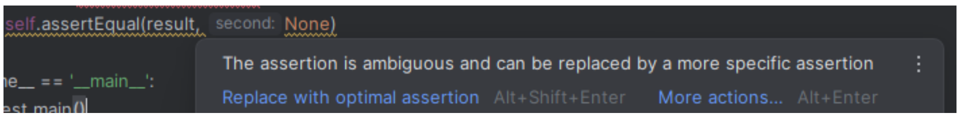
Wrong or Missing Imports. Generated tests sometimes use libraries or functions without importing them, or include unnecessary imports [54]. This can cause execution failures or unnecessary code. The tool inspects import statements to check for undefined or unused references and suggests modifications.

Unused Code. Generated tests might have variables, function definitions, or control flow elements that are never used [54]. For example, assigning a variable without using it. PyTestGuard identifies these unused elements and flags them.

Underused Assertions. Sometimes, in generated unit tests, assertions might not fully use the capabilities of the test framework [6]. For example, as shown in Figure 3.7, `assertE-`

`qual(result, None)` is used, but `assertIsNone(result)` can be used instead. Our tool can assess the relevance of assertions and highlight cases when they can be enhanced with a more effective version.

Although the tool cannot guarantee complete detection, it offers hints that help users refine the generated tests more effectively.



```
self.assertEqual(result, second: None)

ne__ == '__main__':
est main()
```

The assertion is ambiguous and can be replaced by a more specific assertion

Replace with optimal assertion Alt+Shift+Enter
More actions... Alt+Enter

Figure 3.7: Inspection for Underused Assertion in PyTestGuard.

Implementation. To highlight the shortcomings found in LLM-generated unit tests, we reused existing inspections provided by JetBrains, including `PyDocstringTypesInspection`, `PyTypeCheckerInspection`, `PyUnresolvedReferencesInspection`, and `PyUnusedLocalInspection`. For highlighting Underused Assertions, we reused the `SuboptimalAssertTestSmellUnittestInspection` inspection provided by PyNose.

3.3.2 Test smells in LLM-generated tests

To help developers assess the quality of the tests generated by PyTestGuard, we decided to include a test smell detection feature using static code analysis. As we mentioned in Section 2.4, there are currently three tools for detecting test smells in Python: TEMPY [35], pytest-smell [35], and PyNose [94]. Instead of building something from scratch, we chose to integrate one of these existing tools into PyTestGuard.

One of our main requirements was that the tool had to support both test frameworks, pytest and unittest. Because of this, we ruled out pytest-smell, which only supports pytest.

Another important aspect was accuracy in detecting the test smells. TEMPY claims it achieved 100% precision and recall when tested on 365 test cases. Instead, PyNose reports 94% precision and 95.8% recall on 468 test cases. Additionally, in a study by Bodea [35], PyNose was tried on 2,074 test suites (with 1,110 identified as smelly), and the tool was able to detect 84% of the smelly test suites.

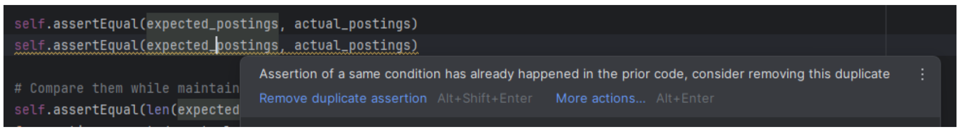
We also considered how easily the smell detection tool could be integrated into PyTestGuard. TEMPY produces reports that show the test smells found, along with the specific method and lines where the smells occur. This makes it possible to parse the reports and display the results in the IDE. PyNose, however, is already integrated with PyCharm, which significantly simplifies our work. By just adding it as a dependency in the `Gradle` build file, we can merge it into our tool. It also detects smells in real time and displays them directly in the editor using inspections.

Given all these requirements, we decided to use PyNose in our tool, as it supports both test frameworks, has good detection accuracy, and is easy to integrate into our tool. Since PyNose can detect 17 different test smells, we decided to focus on a subset of 7 to avoid overwhelming users. These are listed below:

Assertion Roulette. Occurs when a test case has multiple assertions without explanation. It is one of the most common test smells in tests generated by LLMs, along with Magic Number Test [8, 62, 108].
Magic Number Test. Happens when a test method contains undocumented numeric literals. These undocumented numbers reduce readability and maintainability.
Constructor Initialisation. Appears when a test class has a constructor and field initialisations. Since LLMs typically lack full project context, they may introduce unnecessary functions or classes [106].
Duplicate Assert. Happens when there are multiple assertions with the same parameters in a test case.
Empty Test. Occurs when a test method does not contain any executable statements. This is often due to LLMs generating empty or incomplete test cases [32, 85].
Redundant Assertion. Happens when a test method has assertions that always evaluate to true or false. This smell is typical in large or complex projects [62].
Obscure In-Line Setup. Appears when a test case contains ten or more local variables, making the test case hard to read and understand.

As we mentioned above, LLMs sometimes generate incomplete test cases. While the Empty Test smell checks whether a test case contains any executable statements, it does not detect whether a unit test is missing assertions. To address this issue, we implemented a custom inspection class `UnknownTestTestSmellInspection` by extending the `AbstractUniversalTestSmellInspection` class from PyNose. We also registered it as a local inspection[5] in the `plugin.xml` file.

In Figure 3.8, we demonstrate how the test smells are shown in PyTestGuard.



```
self.assertEqual(expected_postings, actual_postings)
self.assertEqual(expected_postings, actual_postings)

# Compare them while maintain
self.assertEqual(len(expected
```

Assertion of a same condition has already happened in the prior code, consider removing this duplicate

Remove duplicate assertion Alt+Shift+Enter
More actions... Alt+Enter

Figure 3.8: Inspection for Duplicate Assert test smell in PyTestGuard.

3.3.3 Inspections

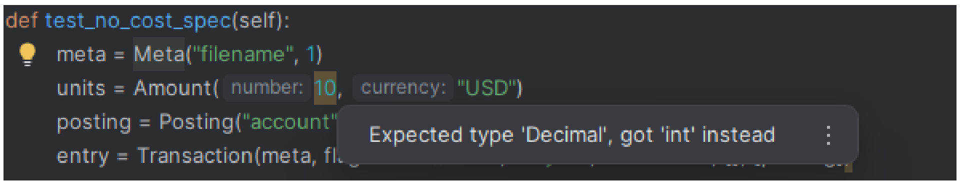
To display both test smell inspections and issues specific to LLM-generated tests, we implemented custom classes that enable inspection support for each test case editor. We created the `TestCaseDocumentCreator` class, which extends the `DocumentCreator` class, to create Python files that store the generated test cases.

We also implemented a `CustomLanguageTextField` class to replace the default implementation, `LanguageTextField`[6], which does not support code inspections. Our custom text field also supports features such as an error stripe, line numbers, and both horizontal and vertical scrollbars to enhance the developer experience.

[5]https://plugins.jetbrains.com/docs/intellij/code-inspections.html
[6]https://plugins.jetbrains.com/docs/intellij/editor-components.html#editortextfield

To simplify inspection configuration in PyCharm, we have provided an XML file containing the inspections supported by PyTestGuard. This file can be imported into PyCharm as an inspection profile[7], making it easier to import and selectively turn inspections on or off.

Finally, to highlight shortcomings found in LLM-generated tests, we used the *Duplicate from server* highlight type (as shown in Figure 3.9), which distinguishes these issues from syntax errors and test smells.



```
def test_no_cost_spec(self):
    meta = Meta("filename", 1)
    units = Amount( number: 10, currency: "USD")
    posting = Posting("account"
    entry = Transaction(meta, fl
```

```
Expected type 'Decimal', got 'int' instead
```

Figure 3.9: Inspection for Wrong Input Type shorcoming in PyTestGuard.

3.4 Execution of Tests

Users can also execute each generated test individually directly from the plugin panel by clicking the "`Run Test`" button at the top of each test panel. When a test is executed, a bold border is shown around the test panel to provide visual feedback to the users. A green border indicates that the test has passed, while a red border means a failing test. The button becomes disabled after being clicked to prevent unnecessary executions. It remains inactive until the user modifies the code, ensuring tests are rerun only when necessary.

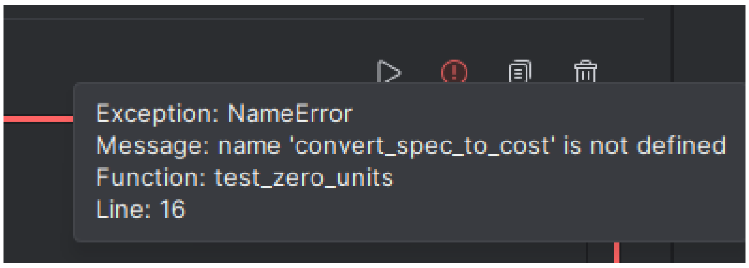
In case of failure, a red icon appears next to the "`Run Test`" button. If a user hovers with the mouse over the icon, a box will pop up with a summarised version of the test's error message. Additionally, clicking the icon allows the user to copy the full exception message. We provide this feature to help users reduce the time needed to read and understand error messages, as eye-tracking studies have revealed that developers dedicate considerable time to interpreting these messages [10].

The summarised exception, as shown in Figure 3.10, includes the exception type, the error message, the function name, and the line number where the error occurred. If the exception originates outside the test file, the name of the corresponding file is also included.

Implementation. When a user runs a test, it triggers a background process[8] that executes a command-line process. The test framework used to generate the tests is also passed as an argument to guarantee correct execution. This process uses `coverage.py`[9] to run the tests and track their coverage. When the process finishes, we extract both the exit code and the execution output message. An exit code of 0 indicates that the test passed successfully, while other codes mean a failure occurred during the execution of the test. This allows us

[7]https://www.jetbrains.com/help/idea/customizing-profiles.html
[8]https://plugins.jetbrains.com/docs/intellij/background-processes.html
[9]https://coverage.readthedocs.io/

to determine the test result programmatically, which can be used for displaying the visual feedback. For the summarised error messages, we parsed and extracted the most important details, such as the exception type, error message, and code location, and passed them as tooltips[10] to the warning icon.



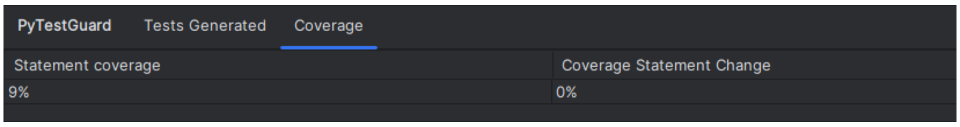
```
Exception: NameError
Message: name 'convert_spec_to_cost' is not defined
Function: test_zero_units
Line: 16
```

Figure 3.10: Summarized execution message shown in case of test failure.

3.5 Coverage of Generated Tests

When a user runs a test case, we also compute its statement coverage, which measures the percentage of code statements executed by the test. The results are displayed in the *Coverage Tab*, as displayed in Figure 3.11. Users can also enable *Coverage Change Mode*, which calculates the statement coverage change, indicating how much the test cases contribute to the existing test suite. This is computed by subtracting the coverage of the test suite without the test case from the coverage of the test suite with the test case. Users can enable this mode in the settings by selecting the *Enable Coverage Change Mode* checkbox and specifying the path of the test suite, as shown in Figure 3.12.

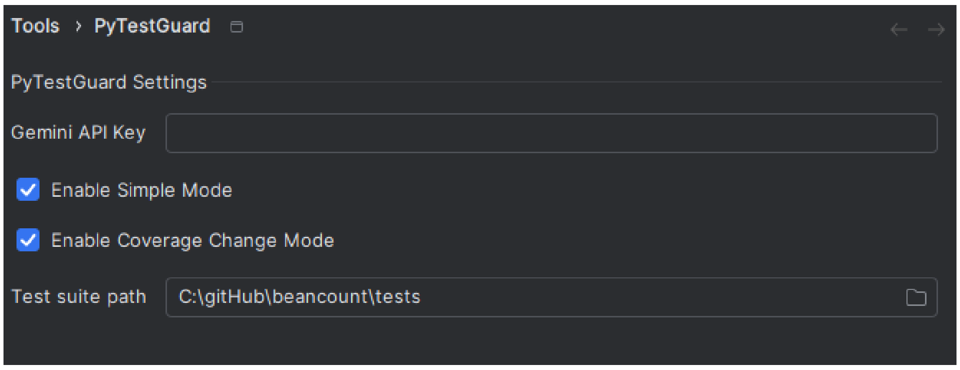
Implementation. As we mentioned in Section 3.4, we use `coverage.py` to run unit tests. When a test is executed, `coverage.py` generates a JSON file containing detailed information about statement coverage. We parse this file to display the results in the *Coverage Tab*. For the statement coverage change, we also use `coverage.py` to run both the original test suite and the test suite that includes the new test case. We then compute the difference in coverage using the data from the corresponding JSON files. Test suites are executed based on the path specified by the developers in the settings.



Statement coverage	Coverage Statement Change
9%	0%

Figure 3.11: Coverage tab in PyTestGuard.

[10]https://plugins.jetbrains.com/docs/intellij/tooltip.html


Tools › PyTestGuard

PyTestGuard Settings

Gemini API Key

[✓] Enable Simple Mode

[✓] Enable Coverage Change Mode

Test suite path: C:\gitHub\beancount\tests

Figure 3.12: Settings page for PyTestGuard.

3.6 Simple Mode for PyTestGuard

PyTestGuard also allows a *Simple Mode*, which focuses only on generating unit tests, without providing advanced features like test execution, code coverage, test smell detection, or inspections. Users can turn on this mode by going to the settings and selecting the *Enable Simple Mode* checkbox, as shown in Figure 3.12. In this mode, the tool uses the default `LanguageTextField` class instead of the custom `CustomLanguageTextField`, which does not support code inspections. We created this version to act as a baseline for our user study, as discussed in Chapter 1, which evaluates the usefulness of PyTestGuard's enhanced features by comparing them against this simplified version of the tool.

Chapter 4

User Study

To answer the research question presented in Chapter 1, we conducted a user study to investigate how IDE tools can be designed to facilitate developers' use and understanding of LLM-generated unit tests within their development workflows. We also examined how these tools impact their motivation and willingness to use large language models for generating unit tests. We asked the participants to complete two tasks using the simple and enhanced versions of PyTestGuard and share their experiences during an interview. In this chapter, we discuss the structure of the interview, the recruitment process, and the project we selected for the tasks.

4.1 Study Overview

The study consisted of six stages designed to guide the research from preparation to evaluation. These stages are shown in Figure 4.1.

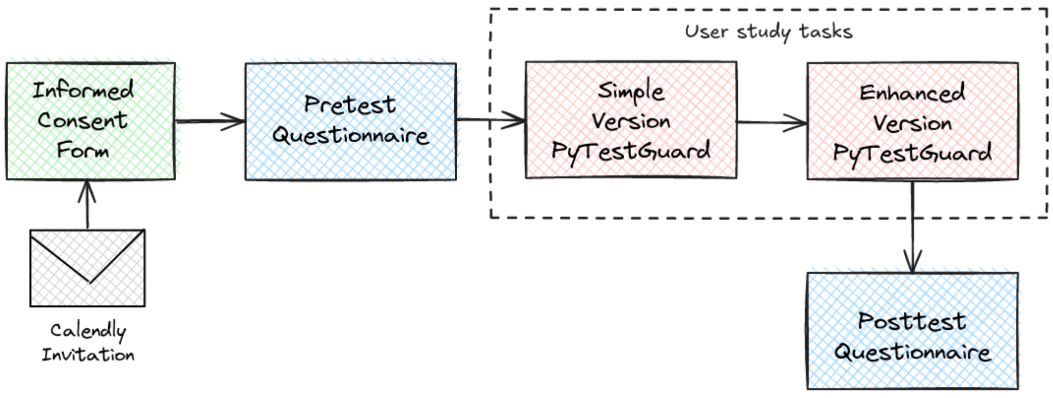


Figure 4.1: Overview of the user study process.

After recruiting the participants via Calendly[1], we sent them an informed consent form via email. This ensured that participants were informed about the study's goal and process,

[1]https://calendly.com/

as well as how their data would be stored and used, before they agreed to participate.

Once participants had completed the informed consent form, we invited them to a remote interview conducted through Microsoft Teams.

Once the interview started, we provided the developers with remote control of the machine, allowing them to try out PyTestGuard without requiring them to install anything on their laptops. We first explained how the experiment would be structured and how they could answer the questions.

We provided them with an already open survey on the remote machine, allowing them to answer the interview questions. We asked them to answer the closed questions (e.g., Likerts) directly on the survey, and for the open-ended questions, they could either type the answers or respond orally, as we recorded the sessions.

During the first 10 minutes of the session, participants completed a pretest questionnaire [42] to collect demographic information, including their general programming background and prior experience using large language models (LLMs) for unit testing purposes. Next, the participants had to complete two tasks where they had to deliver compilable and useful unit tests for specific methods using PyTestGuard. First, we introduced the participants to the first task. They had 10 minutes to generate unit tests using the simple version of our tool, described in Section 3.6. Each participant was assigned to one of two groups and began with one of the two functions shown in Table 4.1, depending on their group assignment. This setup allowed us to randomise the assignment of the function to ensure a fair comparison between the tasks. After generating the tests, participants were asked to review them and, if necessary, fix them within the given time. If we noticed that a developer was struggling with the task, we provided hints to help them continue. Then, we asked participants to reflect on the challenges they encountered while reviewing the generated unit tests and share their thoughts on the quality and correctness of those tests. This first task served as a baseline to evaluate their performance in writing tests without the help of advanced features provided by the enhanced version.

The participants then moved to the second task, where they had again to generate and review unit tests within 10 minutes, this time using the enhanced version of PyTestGuard. This version contained additional features, such as test smell detection. It also provided feedback on issues like syntax errors and hallucinations, helping developers write more effective unit tests. They also had to answer the same questions as in the first task. Before each task, the interviewer explained which features they could use, depending on the version of the tool being used. Throughout the tasks, we took notes of the observations made by the participants or any relevant discussions.

After completing the two tasks, we proceeded to the rest of the user study, which lasted approximately 10 to 20 minutes, depending on the amount of information a participant shared in response to the questions. We asked participants to complete a posttest questionnaire to gather data about their overall experience with the tool, its effectiveness as a testing assistant, and how it compares to other existing automated testing tools. We also asked them whether their opinions about LLMs, given at the start of the study, had changed and if they were more inclined to use LLMs to generate unit tests after using PyTestGuard. Both pretest and posttest questionnaires can be found in Appendices A.2 and A.3.

Table 4.1: Assignment of functions across two tasks (Simple PyTestGuard and Enhanced PyTestGuard) for Group 1 and Group 2. Each group swaps the functions `convert_spec_to_cost` and `postings_by_account` between tasks.

	Task 1 *Simple PyTestGuard*	Task 2 *Enhanced PyTestGuard*
Group 1	convert_spec_to_cost	postings_by_account
Group 2	postings_by_account	convert_spec_to_cost

After the interviews, we sent the participants the session transcripts for their review, allowing them to request the omission of any part before the analysis of the collected data.

4.2 Preliminary Study Preparation

We designed and planned our study by following the guidelines set by the Human Research Ethics Committee[2] (HREC) , as it involves human participants. Before recruiting users, we submitted the study to the HREC for review and obtained approval.

To meet the HREC requirements, we prepared an informed consent form to ensure the participants are aware of the study process. The form included statements on how the interviews would be conducted, what type of data would be collected, how that data would be stored securely, and how it would be used for the research. For each of these points, we explicitly asked for consent from the users. The informed consent, which was sent to the respondents before their interview via a Qualtrics[3] form, can be found in Appendix A.1.

We used convenience sampling [39] to recruit participants for our user study. We sent invitations through posts on LinkedIn and Twitter, as well as via direct messages sent to group chats for master's students and teaching assistants from TU Delft. We also reached out directly to people in our professional networks.

4.3 Project Selection

Since our study consisted of two tasks in which the users were asked to generate unit tests for specific methods, one of our main preparation steps was to select an appropriate Python project to work with. We looked for an open-source project of manageable size, written in clean and readable code, so that participants could understand it within the limited duration of the study.

Data leakage was another factor in our selection process. As Large Language Models (LLMs) may have been trained on publicly available open-source repositories, there is a risk that unit tests for popular projects could already exist in the training data [74]. Since PyTest-Guard depends on Gemini 2.0 Flash, its results may be affected, therefore compromising the study's validity.

[2]https://www.tudelft.nl/over-tu-delft/strategie/integriteitsbeleid/human-research-ethics

[3]https://www.qualtrics.com/

To mitigate this risk, we selected a Python project from The Heap [46], a dataset designed for contamination-free, multilingual evaluation of LLMs in downstream tasks. The Heap handles data contamination in two ways. First, it focuses on code with only non-permissive licenses, such as the GNU General Public License, which prevents its inclusion in training datasets due to licensing restrictions. Second, it applies near and exact deduplication to remove code used in other training datasets, such as The Stack [47].

Therefore, we selected beancount[4] from The Heap. Beancount is a Python project of approximately 45K lines of code, tested using the `unittest` framework. It is a double-entry bookkeeping language that allows financial transactions to be recorded in text files, loaded into memory, analysed through various reports, and accessed via a web interface. Beancount's size and readable code made it feasible for participants to understand it within the limited duration of the study. Additionally, its object-oriented structure was important, as we observed during project selection that LLMs were more inclined to hallucinations in OOP-based code, such as non-existent objects or incorrect function arguments. This made beancount a suitable project for our study since it created scenarios in which PyTestGuard could assist developers in fixing issues generated by LLMs.

4.3.1 Preparation of test cases

To prepare the project for our study, we first forked the original beancount repository to create a separate, working copy that would allow us to safely modify it without affecting the main codebase. Our fork is available on GitHub[5].

As the first step in our experimental setup, we reorganised the existing test cases into a dedicated folder named `tests`. This was necessary to ensure that coverage calculations worked correctly and to make the test locations clear to participants by placing them in a standard structure, separate from the production code.

Next, we selected two methods from the project for our user study's tasks: `convert_cost_to_spec` and `postings_by_account`, which can be found in Appendix B. These methods were chosen because they are well-written and documented, easy to understand, and contain enough logic to make the test generation meaningful. Their clarity and code quality made them appropriate for the short frame of the user study.

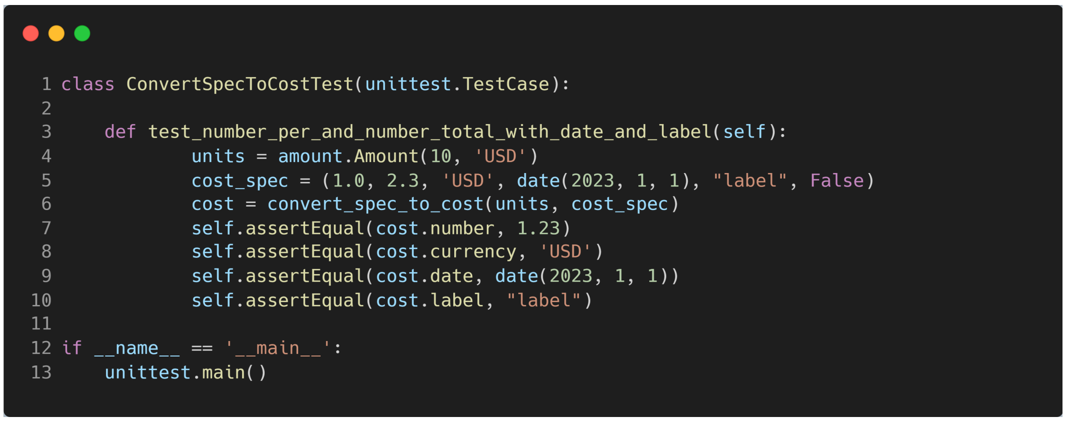
To account for the variability in outputs produced by large language models, we provided participants with a predetermined set of tests instead of asking them to generate the tests themselves during the tasks. We, therefore, guaranteed a consistent baseline for all participants. We created the suite of tests by generating unit tests for each method 10 times using PyTestGuard. Then, we selected 10 tests from the generated pools based on the number of issues and test smells they contained, mentioned in Chapter 3. We prioritised tests with the highest number of shortcomings, guaranteeing participants had chances to identify and fix them during the study. Besides focusing on the test problems during the selection, we also made sure to account for different scenarios and structures in the tests to have a broader and better range of test cases. The selected unit tests were saved in advance and then distributed to all participants through a modified version of PyTestGuard when they

[4]https://github.com/beancount/beancount

[5]https://github.com/nmouman/beancount

were completing the tasks. Since the project uses the unittest framework, we made sure all of our generated tests used it as well to keep everything compatible with the existing setup of the project. For illustration, we provide one example test for each method used in the user study in Figure 4.2.

For our experimental setup, we also removed the original tests of the selected methods so they would not interfere with the outcome of our study. We also precalculated the statement coverage of the test suite of the project, minus the existing tests of the chosen methods. By saving these results, we avoided having to recalculate the coverage during each task, and therefore, helped in saving time and ensuring consistent results across the participants.

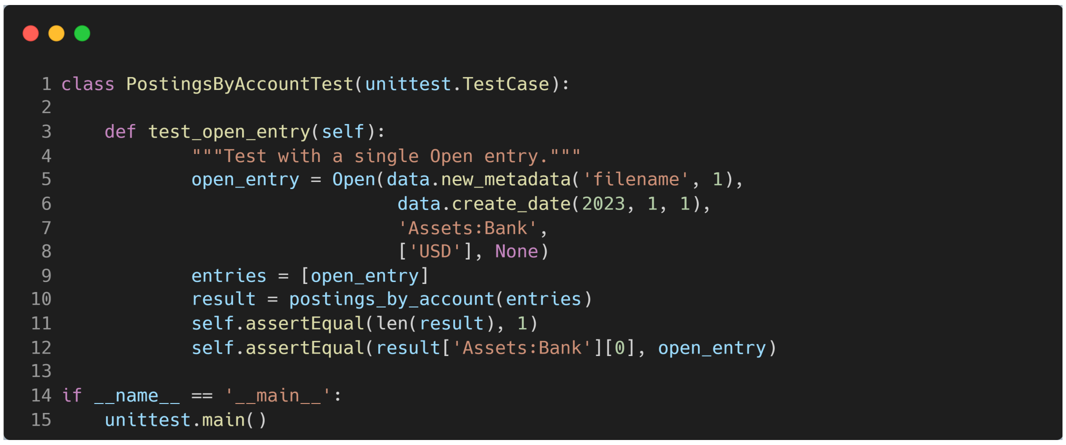


```python
class ConvertSpecToCostTest(unittest.TestCase):

    def test_number_per_and_number_total_with_date_and_label(self):
            units = amount.Amount(10, 'USD')
            cost_spec = (1.0, 2.3, 'USD', date(2023, 1, 1), "label", False)
            cost = convert_spec_to_cost(units, cost_spec)
            self.assertEqual(cost.number, 1.23)
            self.assertEqual(cost.currency, 'USD')
            self.assertEqual(cost.date, date(2023, 1, 1))
            self.assertEqual(cost.label, "label")

if __name__ == '__main__':
    unittest.main()
```

(a) Example test for `convert_spec_to_cost`. Imports omitted for clarity.



```python
class PostingsByAccountTest(unittest.TestCase):

    def test_open_entry(self):
            """Test with a single Open entry."""
            open_entry = Open(data.new_metadata('filename', 1),
                               data.create_date(2023, 1, 1),
                               'Assets:Bank',
                               ['USD'], None)
            entries = [open_entry]
            result = postings_by_account(entries)
            self.assertEqual(len(result), 1)
            self.assertEqual(result['Assets:Bank'][0], open_entry)

if __name__ == '__main__':
    unittest.main()
```

(b) Example test for `postings_by_account`. Imports omitted for clarity.

Figure 4.2: Sample tests used in the user study (line numbers adjusted for missing imports).

4.3.2 Data contamination analysis

Despite selecting a project from the Heap dataset, which minimises the risk of data contamination, it does not eliminate the possibility that the generated tests are just replicas of the original ones. Therefore, to detect any data leakage from the LLM and verify the

Table 4.2: Average cosine similarity scores between original and generated unit tests per function, computed using vector representations and semantic code embeddings.

	Token-based similarity	Semantic-based similarity
convert_spec_to_cost	0.2524	0.6968
postings_by_account	0.3192	0.7008

integrity of our evaluation, we performed a similarity analysis between the original and PyTestGuard-generated unit tests using two code similarity measurements.

Token-based approach. In this technique, we first tokenised the test file to create sequences of code tokens. These tokens are then transformed into vector representations using Term-Inverse Document Frequency (TF-IDF) [72]. TF-IDF is a popular technique in Information Retrieval that converts documents into a term-document matrix based on the importance of each term. Then, we calculated the cosine similarity between the vector representations of the original and generated tests [100]. We chose this technique because, in general, token-based techniques are robust to superficial code changes, such as changes to variable identifiers [104].

Semantic-based approach. We used one model from the CodeT5+ [96] family, a series of open-code large language models that support a wide range of code understanding and generation tasks, to generate semantic embeddings for each test file. The model[6] we used is an encoder-only version with 220 million parameters and a projection layer that produces 256-dimensional embeddings per code. Additionally, CodeT5+ has been fine-tuned for code clone detection, which helps in identifying whether two snippets of code have the same functionality or semantics. We calculated the cosine similarity between these embeddings [31] to compare the original and generated test files. This allows us to detect functional similarity [45] even when the tests are different in syntax or structure.

We calculated the similarity scores pairwise between each original test and every generated test in the complete set, rather than limiting the comparison to the ten unit tests mentioned in Section 4.3.1. This approach guarantees that we catch any potential data contamination across the entire set of generated tests.

The results from the similarity analysis in Table 4.2 show low token-based scores (**0.2524** and **0.3192**) and slightly high semantic-based similarity scores (**0.6968** and **0.7008**) for the functions `convert_cost_to_spec` and `postings_by_account`, respectively. This means that there is no significant token-level overlap, indicating that the model did not simply copy or memorise the original tests. The semantic-based scores indicate that, while the generated tests are similar in functionality to the original ones, they differ in structure.

These findings suggest that the generated unit tests are not the result of direct code duplication but rather that the model is generating tests through generalisation. While we cannot fully guarantee the absence of data contamination, the low token-based and moderate

[6]https://huggingface.co/Salesforce/codet5p-110m-embedding

semantic similarity scores reduce the probability of memorisation or data leakage from the model.

4.4 Interview

For our user study, we conducted semi-structured interviews [29] remotely between June and July 2025, asking participants a mix of closed and open-ended questions. We probed them with further questions when the answer was unclear, and we allowed them to share their thoughts freely during the experiment.

As shown in Table 4.1, we split the participants into two groups, where we exchanged the order of the tests they would use to generate unit tests. We made this design choice to avoid any bias in the ordering of the functions and to have a fair baseline.

We recruited a total of nine participants for the user study. We stopped recruiting when we reached theoretical saturation [73], which meant further interviews were unlikely to yield insights beyond what we had already observed. Most interviews lasted around 45–50 minutes, while two extended to approximately 60 minutes due to participants providing more detailed feedback.

4.4.1 Interview enviroment

We conducted the user study remotely via Microsoft Teams. The participants were provided with control of a remote machine to complete the tasks. The machine was an HP ZBook 15 G5 Laptop with an Intel i7-8750H 2.2 GHz processor and 16 GB of RAM. It had PyCharm 2024.3.5 preinstalled, along with the PyTestGuard and PyNose plugins.

4.4.2 Questionnaire design

For the user study interview, we created a questionnaire in Qualtrics to gather feedback from participants about their overall experience with using LLMs for generating unit tests and their opinions on PyTestGuard. The questionnaire consisted of a mix of multiple-choice questions, open-ended questions, and Likert-scale [53] statements designed to measure participant agreement, frequency of use, and likelihood of future adoption. We structured the questionnaire into six sections, which we describe in detail below:

Demographics. We collected background information on participants to provide context for their interactions with PyTestGuard. It included multiple-choice questions about their years of programming experience, their primary programming languages, their professional role, and how frequently they write unit tests. In one open-ended question, we asked participants to describe the usual challenges they encounter when writing unit tests.

Experience with Large Language Models. Here, we evaluated the participant's familiarity with LLMs and their prior use of automated unit test generation tools. It contained a multiple-choice question asking whether they had ever used automated test generation tools, as well as two Likert-scale questions measuring their likelihood of adopting LLMs in their development workflows and their experience with LLM-generated tests. We also asked what features participants would expect from a tool like PyTestGuard in an open-ended question. The goal was to understand user expectations and their previous experiences with similar tools.

Tasks 1 and 2. Participants used the simple and enhanced versions of PyTestGuard to generate unit tests, and then they were asked to provide feedback based on their experience. We used two Likert-scale questions to measure their trust in the understandability and correctness of the generated tests, as well as their confidence in the unit tests after they had fixed them. In an open-ended question, we asked what challenges they faced during the tasks. We used the same questions for both tasks. The answers from the simple version served as a baseline, while those for the enhanced version allowed us to compare whether the advanced features of the tool addressed the challenges presented in the first task.

Comparison between the two versions of PyTestGuard. This section consisted of two multiple-choice questions that asked participants to say which version of PyTestGuard they preferred in general and which gave them more confidence in the quality of the generated tests. The responses serve to identify whether the advanced features of PyTestGuard improved the user experience overall.

User Experience with Enhanced PyTestGuard. Here, we focused on how the advanced features in PyTestGuard contributed to test quality and ease of use. Two Likert-scale questions evaluate whether PyTestGuard helped participants identify and fix issues in generated tests and whether the information provided (e.g., coverage data, error messages) was clear and helpful. We asked participants an open-ended question to explain how the tool helped them resolve issues like test smells or syntax errors. We also asked developers to rank the most valuable features among those introduced, such as test smell detection, summarised error messages, and coverage metrics. We then ended this section with another open-ended question to suggest additional features. We designed this part of the questionnaire to assess the overall user experience of the participants and which features are most valuable in a tool for generating unit tests.

Assessment of PyTestGuard as a Testing Assistant. In this section, we used four Likert-scale questions to assess whether PyTestGuard increased participants' likelihood of using LLMs for testing, how well it met their expectations, their willingness to continue using it in the future, and the usefulness of integrating it into an IDE. Then, we asked participants to compare PyTestGuard with other tools they had used, such as ChatGPT or GitHub Copilot, and also to provide additional remarks or feedback in two open-ended questions. Here, we aimed to assess the overall practicality and potential adoption of the tool within a development workflow.

The questionnaire used during the user study can be found in Appendix A.

4.4.3 Pilot study

Before conducting the user study, we performed a pilot study with one participant who had experience with both Python and unit testing. The purpose of this pilot was to evaluate the structure of the interview, assess the feasibility and difficulty of the tasks, assess whether each task required any additional information, and estimate the overall duration of the interview.

As van Teijlingen et al. [90] highlight, pilot studies are an important step in identifying potential issues in study designs before data collection begins. In our case, even if we did not find any issues in our design for the experiments, it was still helpful to assess the feasibility and duration of the user study.

During the study design, we estimated that the interview would last between 45 and 60 minutes. However, the pilot study lasted 60 minutes, as we allowed additional time to review each task in detail and get more feedback from the participant. This was not an issue, as we enforced stricter time limits for each task in the actual user study to guarantee that the sessions remained within the estimated duration.

4.4.4 Data collection and analysis

We recorded the complete duration of the interviews, including when participants interacted with the tool and answered the survey questions. We also transcribed the recordings to facilitate the process of analysing the collected data. Eight of the nine participants agreed to the recording of the session, except for one who only agreed to the transcription of the interview. We used the recordings that were available to fix any missing or incorrect parts in the transcriptions.

Afterwards, we used the transcripts to categorise participants' quotes and answers under the corresponding questions in the questionnaire. We then analysed these responses using the coding process from Grounded Theory [24]. Specifically, we conducted open coding on participants' answers, creating codes for which we counted the frequency of occurrence across participants. We used these codes to analyse the results more effectively, rather than examining the raw data directly. For the quantitative survey data, we aggregated the results per question instead.

Additionaly, we analysed the final test files submitted by participants after each task. These were used to compare the tests written with the simple version of PyTestGuard against those written with the enhanced version. We compared the average number of tests, lines, and tokens modified. This comparison was done to evaluate our hypothesis: the simple version would require fewer changes, while the enhanced version would lead to more modifications. This is because participants using the simple version had to identify issues themselves before fixing them, whereas the enhanced version provided quality feedback.

Chapter 5

Results

In this chapter, we present the results of our user study. We start with an overview of the demographics of our participants, including their programming and testing experience. Next, we report on the participants' experience with LLMs for generating unit tests and their expectations of what an LLM-based test generation tool should provide. We then discuss the results of the comparison between the two versions of PyTestGuard during the study tasks, highlighting the challenges participants encountered. After that, we describe the overall experience of developers when using the enhanced version of PyTestGuard, along with the additional features they suggested. Finally, we examine how participants perceived PyTestGuard as a testing assistant and how they compared it with existing tools.

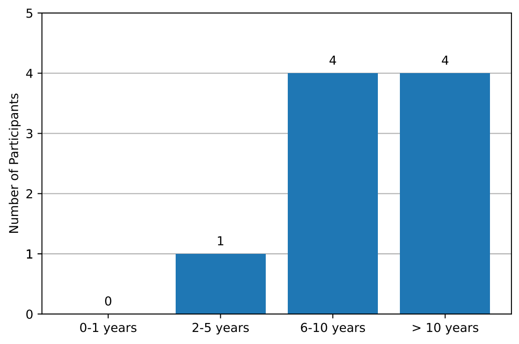
Throughout the results, we report participant observations with a count number, which indicates how many participants agreed with a given statement. We represent these counts with the symbol $\times$. For results based on the Likert scale, we use ratings from 1 to 5 (strongly disagree to strongly agree).

5.1 Participants Demographics

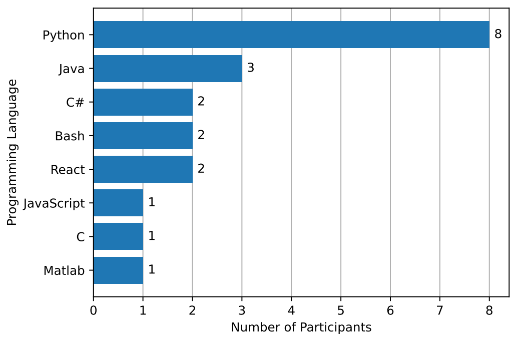
We recruited nine participants for our study, stopping once we felt we had gathered enough data to reach theoretical saturation [73], as additional interviews were unlikely to provide new insights. As an initial step during the interviews, we collected demographic information and details about their experience with unit testing, summarised in Figures 5.1 and 5.2.

As shown in Figure 5.1a, the majority of participants reported having more than six years of programming experience, while one participant indicated between two and five years of experience. Python appeared as the primary programming language used by the participants. Although one participant mentioned that they do not use Python frequently, they still reported some experience with the language. We observe that other languages, such as Java and C#, are also used among the developers, as illustrated in Figure 5.1b.

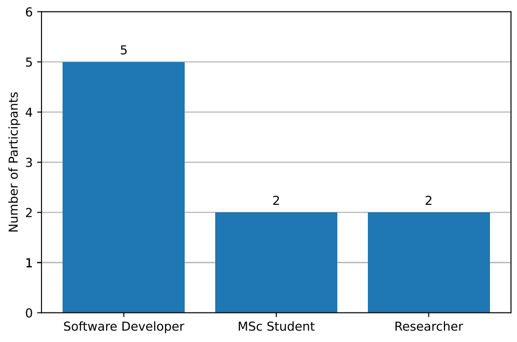
Most participants are currently employed as software developers in the software industry, whereas the remainder are MSc students or researchers, as shown in Figure 5.1c. Among the participants, 5 reported writing unit tests occasionally during their development workflow, while 4 reported writing them frequently (Figure 5.2).



(a) Years of Programming Experience.



(b) Programming Languages.



(c) Professional Role.

Figure 5.1: Demographics of the user study participants.

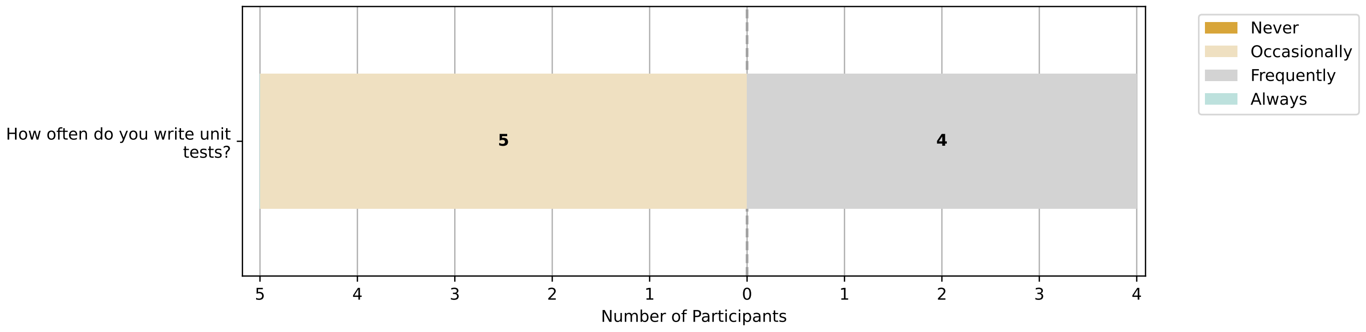


Figure 5.2: Participants' frequency with writing unit tests in their development workflow.

When asked about unit testing, participants shared the main challenges they face when writing unit tests. One issue was remembering both the syntax used in the existing codebase and the correct use of the unit testing framework (3×). Using the correct argument type for a function was also noted as problematic. For example, a developer might pass an `int` to a function that expects a `str` or `long`. Python's type mismatches can cause runtime errors or unexpected results since type checking happens generally during code execution. Similarly, the process of preparing the initial test setup was mentioned as a difficulty (1×).

Participants also described writing unit tests as time-consuming (2×) or, in some cases, unappealing. Participant 2 noted: *"You have to write many unit tests, different options... and finally, when it comes to the programmers. We hate to do it."* As a result, unit tests are sometimes omitted due to time pressure (2×) or delegated to testers, who may implement them differently from the developer's original intention (1×). Additionally, participants have pointed out that testing requires checking for many different conditions (3×), with the risk of missing certain edge cases.

Writing tests can be challenging due to the software's architecture, according to three participants. For example, if a function is dependent on some other components of the code base, it may not be possible to test it on its own. To write an independent unit test, some of these components would have to be modified, or a mock implementation would have to be created.

5.2 Experience with Large Language Models

We collected information about the participants' familiarity with LLMs and their prior use of automated unit test generation tools, as displayed in Figures 5.3 and 5.4.

As seen in Figure 5.3, none of the participants had ever used traditional automated test generation tools that are not LLM-based, like EvoSuite or Pynguin.

When it comes to using LLMs for generating unit tests, participants were overall neutral. About half said they were likely to use them, while the other half said they were unlikely to, as illustrated in Figure 5.4a.

Figure 5.4b shows that four participants had a good experience with LLMs when generating unit tests, while two said their experience was poor. The remaining three had never used LLMs for writing tests. For example, Participant 8, who had never used LLMs, described their view as neutral: *"So far, as I said, I haven't done anything in terms of like generation, so usually I just hand write everything, but yeah, I'm pretty neutral, I would say, if it works, well, then I don't mind using it."*

Participant 6, on the other hand, expressed wariness about using LLMs: *"But yeah, I must admit, I'm very much not on the AI train when it comes to software development. I'm not against it, but I am wary of it."*

Participant 4, who has used LLMs, said they feel guilty when using them, worried that it might reduce their critical thinking: *"Whenever I use an LLM for coding, I feel very guilty for some reason ... Also, I think because like it's been proven by researchers ... if the more you use a large language model ... the more you leave critical thinking behind, and you just accept the answer."*

These comments show that while some participants are open or neutral about using LLMs, others approach them with caution or feel uneasy about counting on them in their development workflow.

We also asked the participants about their expectations for an LLM-based unit test generation tool. Four participants stated that the generated tests should be meaningful and go beyond just covering basic scenarios. As participant 1 explained: *"Well, I would expect the*

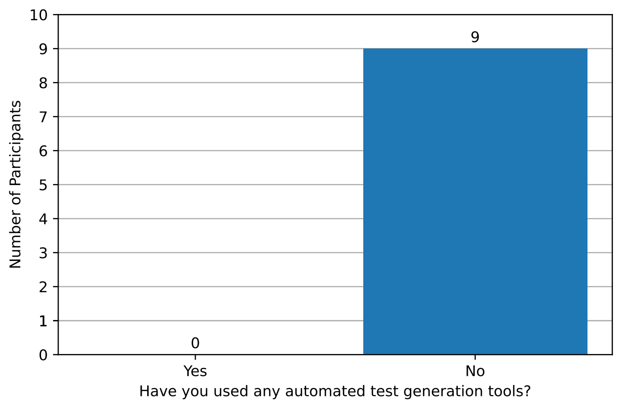
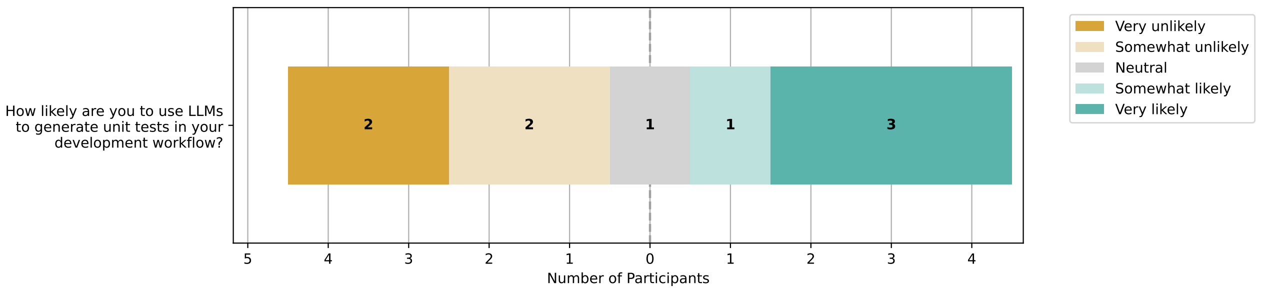
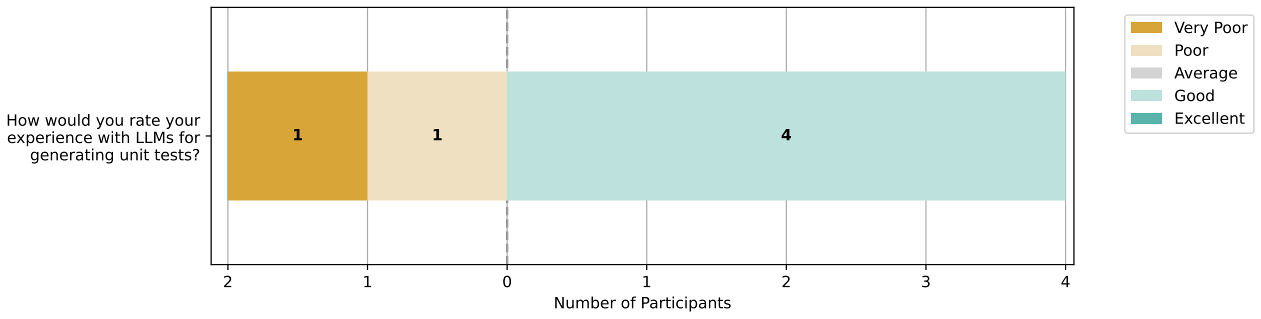


Figure 5.3: Participants' experience with automated testing tools.



(a) Participants's likelihood to use LLMs for generating unit tests



(b) Participants' experience ratings with LLMs for generating unit tests.

Figure 5.4: Overall participants' experience with LLMs for unit testing. Three participants reported having no prior experience with LLMs.

LLM to understand the logic of my code and to be able to create tests that make sense, that can actually help in testing."

Another participant noted that a common issue with LLMs is generating many unnecessary tests. Therefore, the tool should be concise when generating tests. Moreover, a few participants (2×) emphasised the tool's capability to generate tests for uncovered scenarios within the existing test suite or handle edge cases that the developer might overlook. The generated tests should be compilable (1×) and include explanations or inline comments that explain what is being tested (2×).

Three participants added that the tool should be context-aware, meaning it should understand the logic of the code under test and its related files when generating test cases. The tool should be easy to use (2×) and integrated with the workflows and editors of the developers (1×). This integration would reduce the need to switch frequently between environments, as participant 8 stated: *"It would be a bit annoying to, for example, switch tools between like, okay, I use this tool for writing and then I switch tools to write like the tests and then switch back"*.

Another important factor is user autonomy (2×). Developers want to see precisely what changes the tool has made and then decide whether to accept them. Along the same lines, two participants said they would also like the ability to prompt the tool with summary information when generating tests. For example, they could specify a coverage goal or define a good-to-bad weather test case scenario ratio.

Feedback about the wrong use of type in the arguments of a function should also be provided by the tool (1×). One participant indicated they would like the tool to provide explanations when a test is failing. One suggestion is to have indications directly in the editor where the error is happening, similar to how compile-time errors are displayed.

5.3 Comparison Between the Two Versions of PyTestGuard

Midway through the interview, we asked the participants to complete two tasks by using the simple and enhanced versions of PyTestGuard, respectively. We then collected their opinions on the generated tests and the main challenges they faced with both versions, as summarised in Figures 5.5 and 5.6.

In the simple version, we observed that half of the participants did not trust that the unit tests generated by the tool were understandable or correct. In contrast, with the enhanced version, trust increased, as seven participants agreed that the generated tests were understandable and mostly correct. However, one participant still disagreed, stating that although the enhanced version provided feedback on issues, they still had to fix them manually.

Similarly, the majority of participants were not confident that the final tests they wrote using the simple version were runnable, meaningful, and of good quality. With the enhanced version, however, confidence increased, as eight participants reported feeling more confident in writing tests. Participants explained that the feedback on issues and test smells helped them produce tests that were runnable and more meaningful.

At the end of the tasks, we asked participants which version they preferred. As shown in Figure 9, all participants chose the enhanced version, as shown in Figure 5.7. Participant 9 explained: *"I think this one was nicer because it also told me … where and when I did something like a magic number test."*

While participant 8 noted the improvements, they still pointed out limitations: *"I still have run into many of the same issues of course. Like the tests that were generated were still like not usable out of the box. So they still required fixing, but the fixing was a lot easier to do here."*

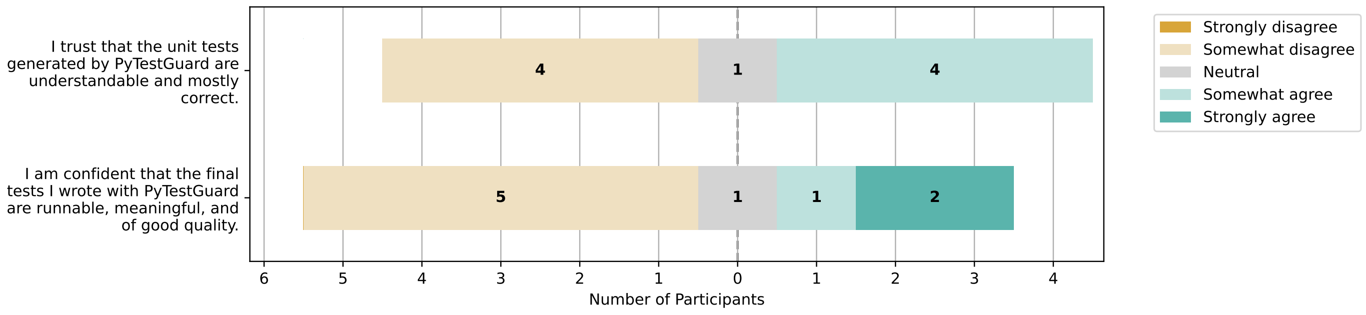


Figure 5.5: Participants' experience with the simple version of PyTestGuard.

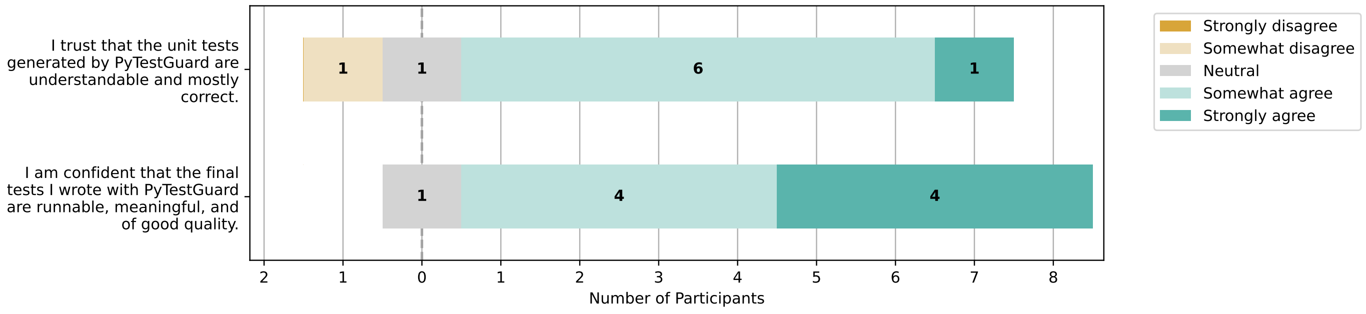


Figure 5.6: Participants' experience with the enhanced version of PyTestGuard.

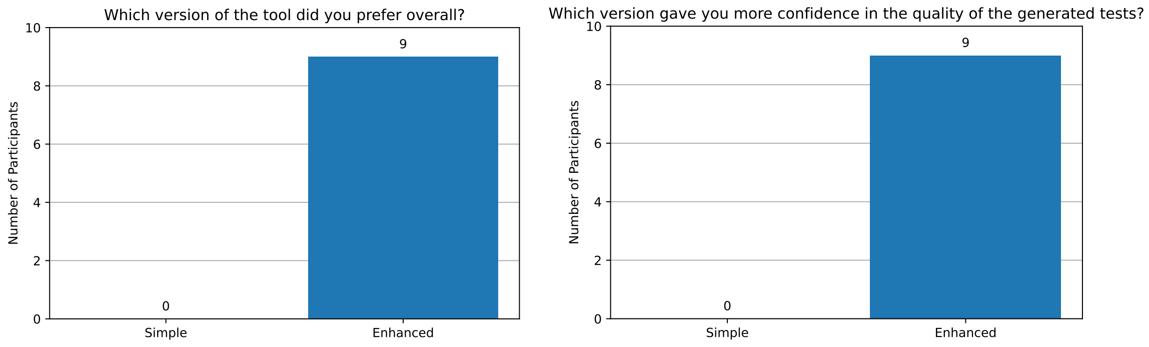


Figure 5.7: Participants' preference of the two versions of PyTestGuard.

Analysis of modifications in generated tests

We also analysed the tests submitted by participants while completing the tasks with both versions of PyTestGuard. Specifically, we compared the average of the number of tests modified and fixed in each version, as well as the average of the total number of lines and tokens modified by the participants. We hypothesised that the simple version would need less editing, while the enhanced version would lead to more editing because of the additional quality feedback. The comparison results are summarised in Table 5.1.

Across both tasks, participants attempted to fix a total of 10 tests. For the function

Table 5.1: Comparison of tests edited with two versions of PyTestGuard with different metrics: Average number of modified tests, Average number of modified lines, Average number of modified tokens.

Metric	**# Modified tests (avg.)**		**# Modified lines (avg.)**		**# Modified tokens (avg.)**	
Version	Simple	Enhanced	Simple	Enhanced	Simple	Enhanced
`convert_spec_to_cost`	2.8	**3.5**	4.6	**6.25**	37.6	**66.5**
`postings_by_account`	1.0	**1.6**	2.0	**7.4**	24.0	**37.6**

`convert_spec_to_cost`, the generated tests had, on average, 17.2 lines of code and 110.6 tokens. For the function `postings_by_account`, the generated tests were larger, averaging 35.9 lines of code and 240.6 tokens, as the tests included more imports.

For `convert_spec_to_cost`, participants modified on average 2.8 tests in the simple version, compared to 3.5 tests in the enhanced version. They edited 4.6 lines of code in the simple version and 6.25 lines in the enhanced version. In terms of tokens, they changed 37.7 in the simple mode and 66.5 in the enhanced version.

A similar trend was observed for `postings_by_account`. In the simple version, participants modified 1 test on average, whereas in the enhanced version, this increased to 1.6 tests. They edited two lines of code in the simple version and 7.4 lines in the enhanced version. Token edits also increased from 24 in the simple mode to 37.6 in the enhanced version.

These results confirm our assumption that the feedback provided in the enhanced version encouraged participants to perform more edits.

Challenges faced while using PyTestGuard

During the execution of both tasks, we noted down and asked the participants about the challenges they faced while writing unit tests with PyTestGuard.

In the simple version of PyTestGuard, developers noticed that the generated tests were sometimes missing the import for the function under test (4×) or included many unused imports (1×). One participant observed that the unit tests were often very similar, differing only in types or values.

During the task, we had to give tips to participants since they struggled to fix hallucinations or incorrect arguments. For example, seven participants struggled with the Decimal object. The generated tests used `ints` or `floats` for numerical values, while the functions expected them wrapped in a `Decimal` object. Because of this, developers had to carefully examine the execution error messages and review the codebase's documentation to understand what was going wrong. Similarly, five participants had difficulties fixing hallucinations, missing arguments, and wrong types since the tool did not provide feedback about these issues.

The main challenge was fixing the hallucination `data.create_date`. The generated tests assumed it would create a date, but it did not exist in the codebase. Participants only discovered that they were supposed to use the `datetime`[1] library after reading the documentation. It was also observed that the simple version was missing a copy button (1×) for moving the tests into the main test suite, as well as any indication about the coverage of the generated tests (1×). Some participants (3×) also mentioned their main difficulties may have come from not fully understanding the code under test.

In the enhanced version of PyTestGuard, some participants (4×) still struggled with hallucinations in the generated tests. For example, for the hallucination `data.create_date`, even though the tool highlighted it as non-existent in the codebase, it did not provide a suggestion on how to fix it, forcing the developers to read the documentation. Another instance was when an assertion expected a negative number, but the generated test used a positive number. Since PyTestGuard does not detect this kind of hallucination, the developers had to spend some time identifying and resolving the issue. Two participants also struggled with assertions using `AssertEqual` instead of `AssertAlmostEqual` for comparing decimal values with multiple places.

The enhanced version also produced tests with missing and unused imports (2×). As in the simple version, some participants (2×) noted their difficulties may have come from not fully understanding the codebase. Participants also found the colour and style of the inspections inconsistent (3×) with PyCharm's regular highlighting. They expected issues to be shown in red, underlined, and wiggly to make them easier to spot. Two participants expected the execution results and the buttons to be better integrated within the IDE. One suggested they should appear in the terminal at the bottom of PyCharm, while the other participant wanted the "Run" button and error icon to move with the developer's view, instead of being fixed at the top of each test box.

While using both versions of PyTestGuard, some participants (2×) pointed out that the generated tests were broken or failing and required manual fixing. Others observed that the generated tests were missing natural language explanations (3×). They suggested explanations could be added as comments inside the tests or at the top of each test box. They would provide information about the assertions and clarify why the tool created those test cases, helping participants better understand the LLM's thought process. While the test method names were descriptive, participants felt explanations would indicate whether a unit test represented a "good or bad weather" scenario.

Another issue across both versions was that PytestGuard generated each test with the whole setup (3×), including the whole test class and imports. The participants stressed that only the test method was necessary, and suggested that a background process could handle a shared set of imports instead.

5.4 User Experience with Enhanced PyTestGuard

After using PyTestGuard, participants shared their opinions on their experience with the enhanced version, as shown in Figure 5.8.

They found the information provided by the tool to be clear and useful and agreed that PyTestGuard helped them identify and fix the issues in the generated tests. All participants

[1]https://docs.python.org/3/library/datetime.html

remarked that the tool supported them with the inspections in identifying the missing arguments, incorrect types, hallucination, and test smells present in the generated tests. Two participants pointed out that it helped with identifying the missing imports and fixing them automatically.

Three participants appreciated that they did not need to copy and paste each test into the main editor to edit and run them, since this could be done directly with the *"Run"* button above each test box.

Participant 6 also liked that it was possible to edit tests while they were running and rerun them immediately, but noted that they would expect such inspections *"to be part of a static analyser that runs in the environment itself, instead of a test generator"*.

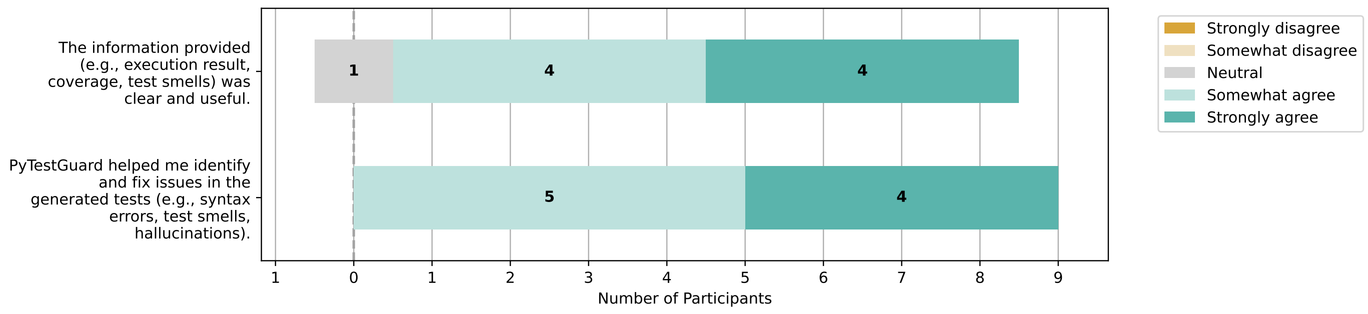


Figure 5.8: Participants' overall experience with PyTestGuard.

We then asked participants to rank PyTestGuard's features by importance from 1 to 6 (first to last), as shown in Figure 5.9. The summarised execution error message was ranked as the most important feature (Ranking = 2.11). Participant 6 explained that they liked it because error messages usually output a lot of information, but the relevant part is often hidden at the end and not always clear.

The test smell detection and inspection for issues in LLM-generated tests were both ranked second (Ranking = 2.56). The visual feedback of the execution results followed in third place (Ranking = 3.89). Some participants mentioned they ranked it lower because they already expected it to be there or would expect it to be integrated with the terminal at the bottom of PyCharm.

Finally, the statement coverage and statement coverage change were ranked fifth (Ranking = 4.67) and sixth (Ranking = 5.22), as some participants (6×) said they did not use these features while completing the tasks.

We then asked the participant which additional features they thought PyTestGuard would need and find most useful.

Three participants expressed the need for a prompt or chat box where they could provide information to the tool when generating test cases. For example, they could prompt the tool to create an additional test case to cover a missing scenario.

Another suggestion was to provide more advanced coverage options (1×). Instead of only showing statement coverage, the tool could provide information about the branch coverage or mutation coverage. Additionally, it would be helpful if the code lines covered by

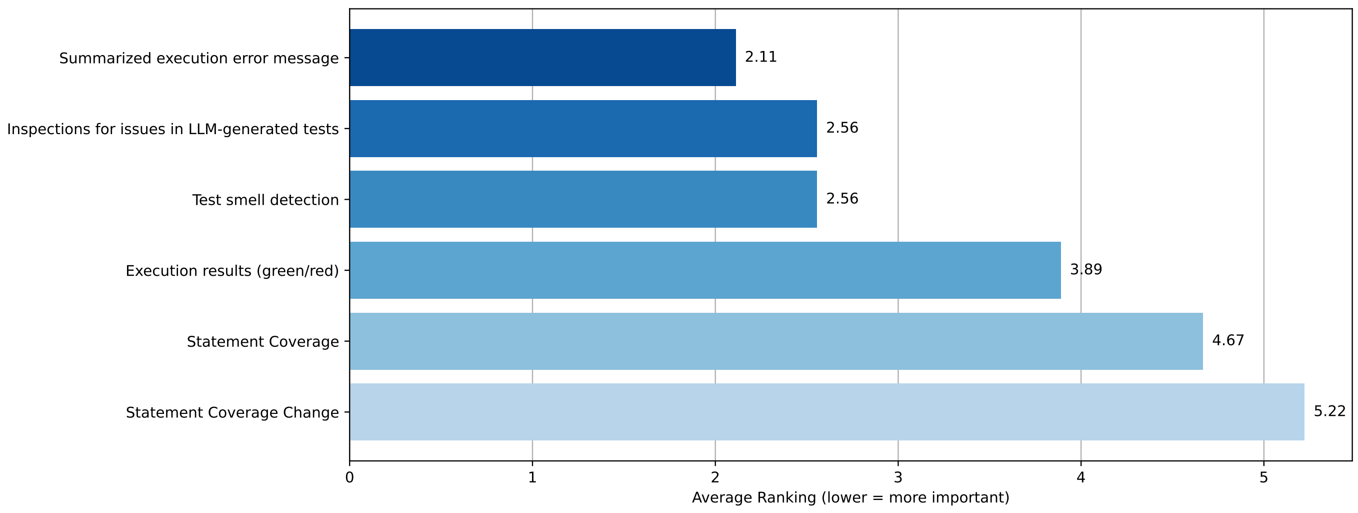


Figure 5.9: Ranking of PyTestGuard's features by importance.

the generated tests were highlighted in the main editor (2×), so the developers can check which parts of the code are being tested.

One participant proposed that PyTestGuard could use an iterative approach, where it would automatically run the tests, check the coverage, and continue generating tests until the coverage is 100%. Alongside this, it was suggested that instead of showing the coverage difference for only a test case, PyTestGuard could show the combined difference for all generated tests (1x). For example, it could indicate that adding three test cases already reaches 80% coverage, while adding all of them together only increases coverage to 82%.

Three participants also expressed that they would like to have more natural language explanations for the generated tests, either as comments or as a short description at the top of each test box. They noted that currently, the tests lack any explanation of what they are doing or why the tool generated them specifically.

It was also commented that once an import is fixed for a test, it would be helpful if it were also fixed automatically in the other test cases (3×). The participants noted that they had to fix the same imports multiple times for different test cases, particularly for the function under test. Similarly, two participants suggested it would be useful if the tool could automatically fix or provide suggestions on how to fix the hallucinations and incorrect types.

Finally, one participant proposed that PyTestGuard could have a button or arrow to insert a generated test method directly into a chosen line in the codebase instead of copying and pasting.

5.5 Assessment of PyTestGuard as a Testing Assistant

At the end of the interviews, participants were asked to assess PyTestGuard as a testing assistant, summarised in Figure 5.10. All participants indicated that they would like to use

PyTestGuard to help them write unit tests in the future. Participant 9 explained that their interest was mainly because *”they really liked the test smell detection”*.

Overall, PyTestGuard met the expectations of most participants as an LLM-based unit test assistant. Participant 5 noted that they expected a chat feature for instructing the LLM since previous tools provided it: *“When a new plugin or a new tool comes around and you try it, you still have the expectation that that tool also offers you similar functionality to what the previous tools offer you ... let’s say you’ve worked with ... GitHub Copilot chat, right? You use it in your IDE, and there you can also ask it questions.”*

Participants agreed that integrating PyTestGuard into an IDE makes sense. As Participant 8 stated: *“Like having a separate tool for writing tests and a separate tool for writing code in general is a deal breaker. Like that would make it so tedious, so integrating it into the IDE is absolutely the right choice.”*

After using PyTestGuard, participants remained neutral about using LLMs to generate unit tests in their workflow. However, there was a noticeable shift since most participants who were previously unlikely to use LLMs moved toward a neutral stance after trying PyTestGuard. One participant stated that their response became neutral simply because they were already very likely to use LLMs, so their opinion did not change significantly. Overall, while there is a slight increase in the likelihood of using LLMs compared to before using PyTestGuard, participants were still wary of LLMs. Participant 9 noted that the complexity of their software might be too much for an LLM: *“Since sometimes there are very complex cases, and I wouldn’t trust an AI to know what to test for, because what we are doing is just too complex for an AI to.”*

Similarly, Participant 5 emphasised the learning benefits of writing tests manually: *“I don’t want to rely too much on Copilot because you know, when you write the test yourself, you also gain more knowledge about the code ... I feel like I understand more about the code if I go about it and define the test myself”*. However, they also expressed that they would still use the LLMs to structure the test suite.

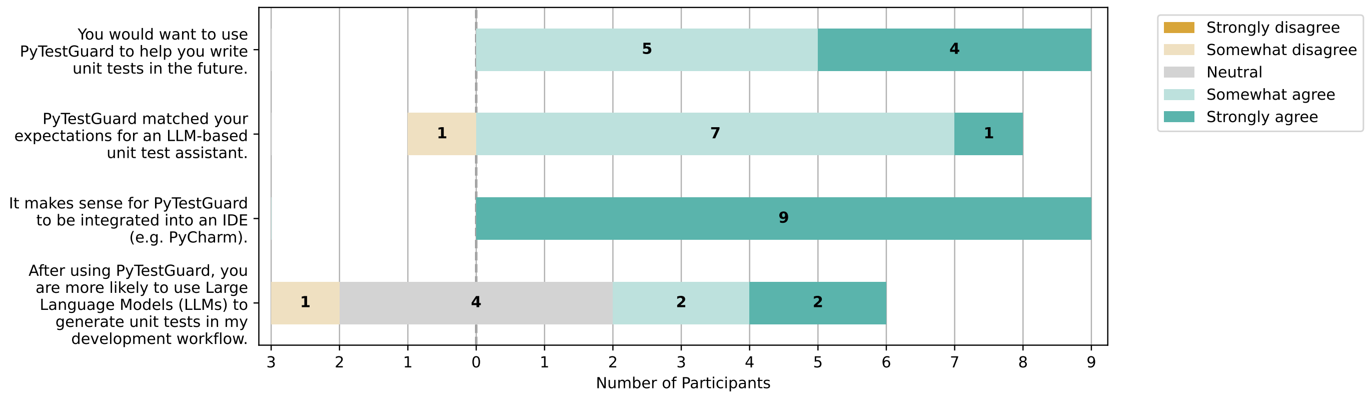


Figure 5.10: Participants’ assessement of PyTestGuard as Testing Assistant.

The participants were then asked to compare PyTestGuard with other tools they have used for generating unit tests, such as ChatGPT and GitHub Copilot.

Three participants pointed out that ChatGPT is currently missing IDE integration and lacks the necessary context to provide feedback about issues in the generated tests, unlike PyTestGuard. It was also noted that with PyTestGuard the developer does not need to switch between different environments when writing unit tests, compared to ChatGPT (1×). However, PyTestGuard does not provide explanations about what the generated tests are doing (1×).

One participant mentioned that PyTestGuard, compared to non-LLM-based tools, produces tests that are concise and have clear names. In contrast, the other tools often generated "gibberish names" and long tests filled with multiple assertions and setup ("arrange") code between them. On the other hand, PyTestGuard cannot guarantee that the generated tests are correct and runnable. Despite this, it was noted that PyTestGuard takes less time to produce tests than the other tools.

Participants also emphasised that PyTestGuard only allows test generation for a specific method (1×). At the same time, GitHub Copilot, for example, supports generation at the level of a specific line or a set of selected lines. However, it was commented that Copilot may rearrange and directly modify the codebase, which can reduce user autonomy compared to PyTestGuard.

Two participants stated that PyTestGuard does not offer a prompt window or chat box where they can specify the type of tests they want, unlike Copilot or JetBrains AI. In addition, one participant noted that in PyTestGuard, they cannot insert the generated tests directly into the code, as in JetBrains AI, but instead have to copy and paste them manually.

Chapter 6

Discussion

In this chapter, we analyse and discuss the results of our user study to answer our research questions. We then explore the potential implications of our findings. Finally, we address the threats to the validity of our study.

6.1 RQ1: What Support Do Developers Expect in an IDE Tool to Assist Them in Using LLM-Generated Unit Tests?

Our findings from Chapter 5 suggest that developers believe IDE support for LLM-generated unit tests to be more than just test generation. The support should instead focus on user trust, ease of understanding, and compatibility with their existing development workflows. Based on their feedback, we categorised the suggested support into five main areas, which are summarised in Table 6.1.

The necessity for *Clarity and Documentation* from the tool was a common theme. Developers highlighted that the generated unit tests should include comments or explanations to facilitate understanding of the cases being covered and tested, thereby improving their comprehensibility [68]. While the descriptive test method names can help users understand the test scenarios, these explanations can also help identify whether any important cases are missing. Explanations could also provide the user with the thought process and rationale behind the LLM when generating tests. Vaithilingam et al. [88] similarly noted the suggestion to include inline comments since some participants in their study had difficulty understanding the code generated by GitHub Copilot. Developers also value explanations for failing tests, as error messages can often be long and unclear [14]. Consistent with our findings in Section 5.4, participants rated the summarised execution error messages in PyTestGuard as the most important feature, indicating that test generation tools should prioritise this form of support.

The participants also emphasised the importance of *Coverage and Diversity* in generated test cases. They noted that the tool should not only cover the base-case scenario but also produce different test cases to ensure higher coverage and cover all possible edge cases. In addition to statement coverage, the tool should support other metrics such as branch and mutation coverage, and provide an overview of the coverage achieved by a selected set of tests, rather than automatically including all generated tests. It was also highlighted the need for the tool to show which parts of the codebase are covered by the tests, potentially through explanations or comments. However, another practical approach would be to highlight the covered lines by the tests directly in the editor, allowing users to identify missing cases visually. Following Brandt et al. [18], the tool could also indicate which other generated tests already cover a given line. This information could help developers decide which set of generated tests to keep to reduce redundancy and improve the quality of the test suite.

Regarding the *Correctness and Reliability* of the generated tests, the participants emphasised that they should be compilable, as they do not want to spend time and effort fixing them manually. One way would be to automatically discard the failing tests and keep only the passing tests. On the other hand, the discarded tests could be testing bug-prone regions that the passing ones do not cover [59]. A solution is to take an iterative approach where it executes the generated tests and prompts the LLM to fix the failing ones by providing the execution information. The tool could keep iteratively creating unit tests until a predefined number of iterations [41, 78] or a desired coverage is achieved [5]. However, we must guarantee that the execution occurs in a sandbox that runs the tests in an isolated environment to prevent potential risks [99]. LLM-generated code can contain vulnerabilities such as filesystem manipulation and access to sensitive information [43, 71].

Effective *Quality Feedback* appears important for developer trust and productivity. Participants emphasised the need for the tool to identify potential hallucinations, such as non-existent objects or incorrect argument types. This would help them to fix the generated tests in a timely and effective manner. While features like test smell detection were valued, the current visual feedback in PyTestGuard was seen as limited. This suggests that future tools should incorporate more proactive feedback, such as automated refactoring or actionable guidance, to help developers write good-quality test suites.

In terms of *User Interaction* design, participants indicated their preference to have the tool integrated within the IDE. The reason was that such integration reduces the time for switching between different environments and allows for generating and adding tests in the same place. Integrating test generation within the IDE would simplify inserting a test at a specific line in the codebase, needing only a single click. Developers also expressed interest in having a prompt box or chat-based interface to guide and personalise test generation [52]. Such interactions could allow users to generate tests for specific scenarios or improve existing ones, giving them greater control over the process. Participants also valued *User Control and Autonomy* [52]. They stressed that the tool should never automatically change or insert code without a clear indication and should provide the option to accept or reject modifications. IDE tools should also support test generation at multiple levels, from individual lines of code to methods, classes, or user-selected code areas, which allows users to have exact control over which parts of the codebase are tested.

Answer to RQ1

Developers expect IDE tools to help them understand and modify the LLM-generated tests. They want clear explanations and comments indicating what each test covers and why they were generated. The tool should generate different and complete tests and provide different types of coverage metrics. It should also allow targeting specific lines, methods, or classes. Tests should be compilable, with failing tests fixed iteratively in a safe sandbox. IDE integration should be seamless, letting users easily add tests and stay in control.

6.2 RQ2: What Challenges Do Developers Face When Using the Simple Version of PyTestGuard Compared to the Enhanced Version?

The results from Section 5.3 suggest that both versions of PyTestGuard influenced how developers completed the tasks in different ways. The simple version forced participants into a guess-run-fix loop, where most of their effort was spent running tests and reading the codebase's documentation to locate issues. Their interaction was focused mainly on finding errors rather than fixing the generated tests. In several cases, when execution error messages were unclear, we had to provide hints to help them complete the task. This was noticeable in cases where there were missing imports, type mismatches, and hallucinated objects, as the simple version did not provide feedback. For example, we observed this when a function expected a `Decimal` object, but the generated tests used an `int` or float instead. The error message only indicated that the value was not a valid numerical value, therefore, the participants had to review the documentation or get hints to understand the issue.

This was not the case for the enhanced version, which informed the developers that the function was expecting a `Decimal` object as an argument. Here, the developers focused more on fixing issues rather than discovering them, since feedback was provided on where they occurred. The enhanced version helped with fixing wrong types, missing arguments and imports. However, some hallucinations still caused problems, as they could only be discovered after execution. For example, one case occurred when an assertion used positive numbers instead of negative ones. Even though developers located issues faster in the enhanced version, they sometimes still had to consult the documentation. We observed this in the hallucinations of non-existent objects, as the tool only indicates that the object did not exist without providing any suggestions on how to resolve the issue.

The results of the analysis of modifications in generated tests in both versions in Section 5.3 also support the point that the simple version led the participants to focus more on locating issues, and the enhanced version, instead, on focusing on fixing the generated tests. This is because overall, the enhanced version led to more modifications in the generated tests compared to the simple version.

Table 6.1: List of suggested features to include in future IDE tools for automated test generation and their descriptions.

Category	Feature	Description
Clarity & Documentation	Explanations/Comments in Tests	Generated tests should include inline comments or explanations that describe what is being tested.
	Failing Test Explanation	Summarizes errors and explains failing tests clearly, which reduces time spent interpreting long error message.
Coverage & Diversity	Coverage Metrics	Provides statement, branch, mutation coverage, and overall coverage of selected test sets.
	Coverage Indication	Highlights which parts of the code are covered by tests, either in editor or via comments.
	Different Test Cases	Generates base cases and edge cases by covering different scenarios, which ensures higher code coverage.
Correctness & Reliability	Compilable Tests	Ensures generated tests compile without manual fixing.
	Iterative Test Fixing	Executes generated tests in a sandbox and prompts LLM to fix failing tests iteratively.
	Sandboxed Execution	Runs generated unit tests in an isolated environment to prevent potential risks from unsafe code in LLM-generated tests.
Quality Feedback	Hallucination Detection	Identifies non-existent objects or incorrect argument types in tests.
	Test Smell Detection	Highlights poor programming practices in generated tests.
	Automatic Refactoring Suggestions	Provides suggestions to fix issues like type errors or missing objects.
User Interaction & Control	Interactive Feedback	Allows users to provide feedback or customise generated tests through a prompt box or chat-based interface.
	Level-Specific Test Generation	Supports generating tests for specific lines, methods, classes, or user-selected areas.
	User Control	Tool never modifies code automatically and changes require explicit acceptance.

Answer to RQ2

Our findings suggest that in the simple version, participants spent most of their time locating errors. This was mainly due to their struggles with missing imports, type mismatches, and hallucinated objects. They had to review documentation and error messages, or rely on hints provided by the interviewer to fix these issues. Instead, the enhanced version provided feedback on these issues, which allowed developers to focus more on fixing errors rather than discovering them. However, participants still struggled with some of the hallucinations (e.g., negative numbers in assertions), which were not detected by the tool. Overall, the enhanced version led to more modifications in the generated tests, which indicates a shift from locating errors to fixing them.

6.3 RQ3: How Does PyTestGuard Affect Developers' Willingness to Use LLM-Generated Unit Tests?

We analyse the results from Sections 5.2 and 5.5 and compare the likelihood of participants using LLMs for generating unit tests in their development workflows, before and after trying PyTestGuard.

Before using PyTestGuard, two participants said they were very unlikely and another two were somewhat unlikely to use LLMs. One participant noted they were neutral about it, while the remaining four participants (about half) were likely to use them. After using PyTestGuard, we can see some improvements in terms of likelihood. Most participants reported that they were no longer very or somewhat unlikely to use LLMs. Only one participant still leaned toward somewhat unlikely. Four participants said they were now more likely to use LLMs, while the rest remained neutral.

The quality feedback in PyTestGuard was found useful, as it helped identify issues in generated tests. However, participants noted that the tool is not yet reliable enough for day-to-day use, as it often requires manual fixes to the tests. We also notice that some participants doubted whether LLMs could handle the complexity of the software they work on [98]. As Participant 6 noted: *"The more complex things get, the more the LLMs struggle with getting the right context in place."*

Participants also emphasised that they still prefer writing tests manually because it helps them better understand their code and maintain their critical thinking. Some even mentioned feeling "guilty" when relying on models for generating unit tests. This corresponds with the results identified by Weisz et al. [98], who noted that developers are concerned that AI assistants could result in a loss of skills. As our participants also noted, LLMs can still help generate boilerplate code for them.

Overall, these insights suggest that PyTestGuard shows much promise for the future, but developers are more likely to adopt it if the tool provides reliable support that fits into their existing workflows. Making improvements, such as clearer explanations or guaranteeing the generated tests always compile, could encourage more developers to use LLMs for generating unit tests.

Answer to RQ3

PyTestGuard slightly increased participants' willingness to use LLMs for unit test generation. Before trying it, almost half were unlikely, one was uncertain, and the rest were likely to use them. After using it, only one participant was unlikely, some became "more likely," while others stayed uncertain. Participants valued feedback on test quality issues but still saw the tool as untrustworthy since tests often needed manual fixing. Some stated they would use it more if it guaranteed compilable tests and clearer explanations.

6.4 Implications

In this section, we discuss the implications of the results from our user study for three groups: practitioners, researchers and tool designers.

Implications for Practitioners. While LLM-generated tests can reduce the time developers spend on writing tests, they often need to be reviewed manually to ensure they are correct. In fact, we observed this aspect when developers used PyTestGuard to write unit tests. Most of their time and effort was spent locating errors in the simple version of PyTestGuard and fixing issues in the enhanced version to deliver correct tests. Hence, practitioners need to find a balance between saving time and ensuring the quality of generated unit tests.

Developers should treat LLM-generated tests as a helpful starting point instead of a replacement for their own test design. While LLMs can generate basic and edge case scenarios, developers should still review the generated tests to make sure they cover the expected system behaviour. While PyTestGuard highlights issues like missing imports, hallucinations, and test smells, we noticed in our study that developers still need to review the generated tests carefully. There may be other problems that the tool misses, such as using negative numbers instead of positive ones in assertions. Therefore, we stress that developers cannot rely only on the issues highlighted by the tool, but they also need to review the generated tests for non-detectable issues.

Practitioners should also prioritise using tools that provide clear and actionable feedback. In our study, developers found the unit test generated by the enhanced version of PyTestGuard to be more understandable and correct compared to the simple version. Participants also expressed that they felt more confident about the quality of the unit tests they delivered with the enhanced version. This suggests that tools that provide indications about test failures, syntax errors, or suggest fixes make the generated test more reliable. Additionally, it helps developers trust the tests more and integrate them into their development workflow.

Implications for Researchers. Our findings suggest that developers prefer the LLM-generated tests to include explanations or inline comments. These can help them to understand which scenarios the tests are covering and the thought process used by the LLMs in generating the tests. Future research could evaluate how different types of explanations influence the trust of developers and the adoption of LLM-generated unit tests.

Developers also expressed their preference to interact with the LLM to refine the generated test, provide feedback or guide it in achieving better coverage. Researchers should investigate how to design practical interactive approaches, such as prompt boxes and chat-based interfaces and examine how developers react to them in practice. A helpful strategy would be to allow these approaches also to take project-specific context into account smartly. For example, predefined prompts with detailed information about the codebase could be used along with the feedback provided by the developers.

PyTestGuard features, such as test smell detection and inspections for issues like hallucinations and incorrect types in function calls, helped the developers to fix the generated tests. However, there is still some wariness around LLMs since developers still have to fix the tests manually. Developers might be more willing to use LLMs if the tools automatically fix them. Future work should therefore investigate ways to either generate compilable tests directly or provide meaningful suggestions that guide developers in fixing these issues more efficiently.

Our study results are based on developers generating and fixing unit tests for an example project. Future research should explore how developers perceive and use PyTestGuard in real-world settings, particularly with larger-scale software systems that rely on non-common or domain-specific technologies [98].

Implications for Tool Designers. During our user study, developers suggested that PyTestGuard could, in the future, provide explanations for each generated test, explaining the scenarios they are covering or the reasoning behind their creation. They also expressed interest in interacting with the LLM to give feedback or refine the test cases. Future tools should therefore include clear explanations or inline comments for generated unit tests and offer interactive features, such as prompt boxes or chat-based interfaces, to allow developers to guide and improve the test generation process.

The study revealed some problems with IDE integration. Developers expressed interest in more consistent highlights in PyTestGuard with the existing highlighting in PyCharm. They also wished for a better placement of the error messages, which could be provided as terminal-based feedback instead of simple icons with the error message as a tooltip. Future work could explore how to achieve better integration into other IDEs and improve other usability features, such as the highlight types for the issues present in the generated tests.

6.5 Threats to Validity

In this section, we discuss different factors that could threaten the validity of our user study results.

Internal validity. We compared two versions of PyTestGuard that differed only in the presence or absence of specific features. All other settings, including the LLM configuration and implementation details, were kept the same. Consequently, we ensured that any differences we observed were likely because of the features themselves, rather than other aspects.

Additionally, to reduce the risk of learning effects from the developers completing the tasks using each PyTestGuard version, we provided a different set of tests for each task. We also split participants into two groups and switched the order of the tests they used to generate unit tests. This design choice helped us to avoid bias from task ordering and guaranteed a fair baseline for comparison.

Another possible threat is that LLMs may memorise training data, which could lead to data contamination. Therefore, we performed a data contamination analysis to verify that the model did not generate tests already present in the test suite of the project used in our study but provided new test cases.

Confirmability. We based our findings on the interview data by relying only on participants' transcripts and systematically analysing the quotes and codes that came from them. This approach helps to ensure that the results reflect what participants actually said rather than our own interpretations. We also include the list of codes and participants' answers in the replication package [61].

Respondent Bias. Since participants knew that we developed the tool, it's possible that participants' feedback was affected by an attempt to provide positive responses. To reduce this, we let them use PyTestGuard freely and share their opinions during the interviews. We actually received negative feedback about missing comments and explanations in the generated tests, as well as other usability issues. This shows that participants felt comfortable expressing their honest opinions about the tool, suggesting that we have mitigated this threat.

External validity. Our results might not be entirely generalisable for several reasons. Since we worked with beancount as our example project for our user study, the results could be different on projects with other characteristics. For example, projects that are not object-based might lead to fewer hallucinations in the generated tests.

PyTestGuard also relies on Gemini as the LLM and uses Chain-of-Thought as a prompting technique. A different model might yield different results since other LLMs may have different training data or architecture. The same applies to different prompting approaches, which could provide more or less context to the LLM, and therefore lead to different outcomes. Our focus was mainly on Python, which means our findings might not apply to statically typed languages such as Java.

Another factor is that we used convenience sampling to recruit participants, therefefore, our results mainly come from our professional networks and people interested in LLM-gerated unit tests. As explained in Section 5.1, we recruited a diverse group with different programming experience and backgrounds (including developers, researchers, and MSc students). However, the group is small (9 participants), and we currently lack input from developers with 0–1 years of experience. A larger and more balanced participant pool could lead to different results.

Construct validity. One threat to the construct validity of our study is that our results are based on using only PyCharm as the IDE. Some participants may have been disadvantaged if they typically use other IDEs in their daily work or had no prior experience with PyCharm. This could have affected both their performance on the tasks and their opinion of PyTestGuard. We tried to mitigate this threat by explaining how to use the tool and the user study process. We also provided explanations during the tasks whenever something was unclear.

Another possible threat is that participants were unfamiliar with the project used in the study. This may have influenced their opinions on PyTestGuard since they had to spend time understanding the code under test instead of evaluating the feedback provided by the tool.

Reliability and Reproducibility. A potential threat to reproducibility may arise from the ongoing development of large language models such as Gemini. Because these models are continuously updated and improved, exact replication of the results may not be possible in the future. Particularly, PyTestGuard relies on Gemini-2.0 Flash to generate unit tests. To address this, we provide a replication package [61] that includes all code and data necessary to reproduce our study. This package also contains the set of tests that developers used to complete tasks during our study.

To ensure the reliability of our results, we generated approximately 100 unit tests in 10 iterations for each task, and then selected 10 tests per task for our user study. By doing so, we reduce the random nature of the LLM to generate different tests per participant and provide each developer the same baseline to ensure there is no bias. This approach mitigates the non-deterministic nature of LLMs when generating unit tests, providing all developers with a consistent baseline.

Chapter 7

Conclusions and Future Work

In this chapter, we conclude our thesis by summarising the research approach and the results from our user study. We also discuss potential suggestions for PyTestGuard to consider for future work.

7.1 Conclusions

The goal of this thesis was to investigate the support developers expect from IDE tools for automated test generation using Large Language Models (LLMs). We designed PyTestGuard, a plugin for PyCharm that goes beyond generating unit tests for Python by providing feedback on the quality of the tests. The tool points out issues such as test smells, missing or incorrect arguments in function calls, and highlights hallucinations like non-existent objects and imports. It also allows developers to edit and run the tests directly and shows information about statement coverage.

We evaluated PyTestGuard through a user study with nine participants. They had to complete two tasks using two versions of the tool: a simple version, which only generated unit tests, and an enhanced version, which also included the advanced features. From the study, we found that participants mainly struggled with fixing hallucinations. Even though PyTestGuard identified them, it did not provide suggestions on how to fix them. Participants also wanted more explanations in the tests, either at the top of each test or as inline comments, to clarify what scenarios were being tested and whether important cases were missing.

It was also clear that developers wanted ways to interact with the LLM, for example, through a prompt box or a chat interface, so they could give input on how to improve or correct the generated tests. At the same time, participants showed some caution about using LLMs for unit test generation. They worried that relying too much on them might reduce their own critical thinking and skills. They also noted that the generated tests often needed manual fixing because some were broken or failing. One suggestion from the participants was to use an iterative approach, where the LLM is prompted with execution errors until the tests pass or a limit is reached.

Overall, PyTestGuard shows the potential of combining test generation and feedback in

IDEs with the help of LLMs. However, the results also highlight that developers want more than just the detection of the issues. They expect explanations, suggestions, interaction, and ways to improve failing tests. Future tools should support developers in these areas, while making sure LLMs act as a helpful assistant rather than a replacement.

7.2 Future Work

While PyTestGuard provides support to developers and was found helpful by our participants, there are still some improvements that could be made in the future.

One improvement would be to implement the suggestions made by the participants during our user study. For example, adding automatic program repair for tests with syntax errors or hallucinations, such as using objects that do not exist, or using the wrong argument types. To handle this, we could combine LLMs with more traditional tools like static analysis or type checking. The goal would be to propose fixes directly to the developer using suggestions or actions within the IDE.

Another option would be to automatically fix the generated unit tests by using an iterative approach. PyTestGuard would continuously prompt the LLM with the error execution messages until a predefined number of iterations or the tests pass. This would need to be done in a secure sandbox environment to prevent the generated code from introducing any vulnerability into the user's system.

PyTestGuard currently only works with Python, but we could extend the support to other programming languages, such as C#. Current LLM-based tools have mainly focused on Java and Python, which leaves other programming languages underrepresented [93]. Similarly, we could extend support to other popular IDEs, such as VS Code, other than PyCharm.

Additionally, we could explore the performance of LLMs on other types of testing, such as integration testing, to investigate whether they also introduce issues like hallucinations or whether they achieve better results.

PyTestGuard only supports method test generation, hence, we could consider expanding to other levels, such as line and class levels, as well as specific code regions. We could also expand PyTestGuard to support different models, such as GPT-4, other than Gemini. We could also allow the use of local models since some companies might not allow developers to use third-party AI tools [1].

Finally, we should consider investigating different prompt techniques and how they affect the quality of generated tests. One possible approach would be retrieval-augmented generation (RAG) [34]. It is a context retrieval technique that retrieves relevant context from the project itself, using a vector database that stores semantically similar pieces of code and documentation. Using this context to prompt the LLMs could lead to more reliable generated tests.

Bibliography

[1] Software developers statistics 2024 - state of developer ecosystem report, 2024. URL `https://www.jetbrains.com/lp/devecosystem-2024/`.

[2] Anisha Agarwal, Aaron Chan, Shubham Chandel, Jinu Jang, Shaun Miller, Roshanak Zilouchian Moghaddam, Yevhen Mohylevskyy, Neel Sundaresan, and Michele Tufano. Copilot evaluation harness: Evaluating llm-guided software programming, 2024.

[3] Toufique Ahmed and Premkumar Devanbu. Few-shot training llms for project-specific code-summarization. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450394758. doi: 10.1145/3551349.3559555. URL `https://doi.org/10.1145/3551349.3559555`.

[4] Wajdi Aljedaani, Anthony Peruma, Ahmed Aljohani, Mazen Alotaibi, Mohamed Wiem Mkaouer, Ali Ouni, Christian D. Newman, Abdullatif Ghallab, and Stephanie Ludi. Test smell detection tools: A systematic mapping study. In *Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering*, EASE '21, page 170–180, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450390538. doi: 10.1145/3463274.3463335. URL `https://doi.org/10.1145/3463274.3463335`.

[5] Juan Altmayer Pizzorno and Emery D. Berger. Coverup: Effective high coverage test generation for python. *Proc. ACM Softw. Eng.*, 2(FSE), June 2025. doi: 10.1145/3729398. URL `https://doi.org/10.1145/3729398`.

[6] Victor Alves, Carla Bezerra, and Ivan Machado. An empirical study on the detection of test smells in test codes generated by github copilot. In *Anais Estendidos do XV Congresso Brasileiro de Software: Teoria e Prática*, pages 69–78, Porto Alegre, RS, Brasil, 2024. SBC. doi: 10.5753/cbsoft_estendido.2024.4102. URL `https://sol.sbc.org.br/index.php/cbsoft_estendido/article/view/30247`.

[7] Victor Alves, Cristiano Santos, Carla Bezerra, and Ivan Machado. Detecting test smells in python test code generated by llm: An empirical study with github copilot. In *Anais do XXXVIII Simpósio Brasileiro de Engenharia de Software*, pages 581–587, Porto Alegre, RS, Brasil, 2024. SBC. doi: 10.5753/sbes.2024.3561. URL https://sol.sbc.org.br/index.php/sbes/article/view/30398.

[8] Victor Anthony Alves, Carla Ilane Moreira Bezerra, Ivan Machado, Larissa Soares, T'assio Virg'inio, and Publio Silva. Quality assessment of python tests generated by large language models. *ArXiv*, abs/2506.14297, 2025. URL https://api.semanticscholar.org/CorpusID:279410322.

[9] Andrea Arcuri, José Campos, and Gordon Fraser. Unit test generation during software development: Evosuite plugins for maven, intellij and jenkins. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 401–408, 2016. doi: 10.1109/ICST.2016.44.

[10] Titus Barik, Justin Smith, Kevin Lubick, Elisabeth Holmes, Jing Feng, Emerson Murphy-Hill, and Chris Parnin. Do developers read compiler error messages? In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, page 575–585. IEEE Press, 2017. ISBN 9781538638682. doi: 10.1109/ICSE.2017.59. URL https://doi.org/10.1109/ICSE.2017.59.

[11] Manaal Basha and Gema Rodríguez-Pérez. Trust, transparency, and adoption in generative ai for software engineering: Insights from twitter discourse. *Information and Software Technology*, 186:107804, 2025. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2025.107804. URL https://www.sciencedirect.com/science/article/pii/S0950584925001430.

[12] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and David Binkley. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *2012 28th IEEE international conference on software maintenance (ICSM)*, pages 56–65. IEEE, 2012.

[13] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. Are test smells really harmful? an empirical study. *Empirical Software Engineering*, 20(4):1052–1094, 2015.

[14] Brett A. Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J. Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, Janice L. Pearce, and James Prather. Compiler error messages considered unhelpful: The landscape of text-based programming error message research. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*, ITiCSE-WGR '19, page 177–210, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450375672. doi: 10.1145/3344429.3372508. URL https://doi.org/10.1145/3344429.3372508.

[15] Joel Becker, Nate Rush, Elizabeth Barnes, and David Rein. Measuring the impact of early-2025 ai on experienced open-source developer productivity, 2025. URL `https://arxiv.org/abs/2507.09089`.

[16] Robin Beer, Alexander Feix, Tim Guttzeit, Tamara Muras, Vincent Müller, Maurice Rauscher, Florian Schäffler, and Welf Löwe. Examination of code generated by large language models, 2024.

[17] Shreya Bhatia, Tarushi Gandhi, Dhruv Kumar, and Pankaj Jalote. Unit test generation using generative ai : A comparative performance analysis of autogeneration tools. LLM4Code '24, page 54–61, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400705793. doi: 10.1145/3643795.3648396. URL `https://doi.org/10.1145/3643795.3648396`.

[18] Carolin Brandt and Andy Zaidman. How does this new developer test fit in? a visualization to understand amplified test cases. In *2022 Working Conference on Software Visualization (VISSOFT)*, pages 17–28, 2022. doi: 10.1109/VISSOFT55257.2022.00011.

[19] Carolin Brandt and Andy Zaidman. Developer-centric test amplification. *Empirical Software Engineering*, 27(4), 2022. ISSN 1382-3256. doi: 10.1007/s10664-021-10094-2.

[20] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA, 2020. Curran Associates Inc. ISBN 9781713829546.

[21] Denivan Campos, Larissa Rocha, and Ivan Machado. Developers perception on the severity of test smells: an empirical study. *arXiv preprint arXiv:2107.13902*, 2021.

[22] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage,

Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021. URL `https://arxiv.org/abs/2107.03374`.

[23] Xiang Chen, Chaoyang Gao, Chunyang Chen, Guangbei Zhang, and Yong Liu. An empirical study on challenges for llm application developers. *ACM Trans. Softw. Eng. Methodol.*, 34(7), August 2025. ISSN 1049-331X. doi: 10.1145/3715007. URL `https://doi.org/10.1145/3715007`.

[24] Juliet M Corbin and Anselm Strauss. Grounded theory research: Procedures, canons, and evaluative criteria. *Qualitative sociology*, 13(1):3–21, 1990.

[25] Ermira Daka and Gordon Fraser. A survey on unit testing practices and problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 201–211. IEEE, 2014.

[26] Benjamin Danglot, Oscar Vera-Perez, Zhongxing Yu, Andy Zaidman, Martin Monperrus, and Benoit Baudry. A snowballing literature study on test amplification. *Journal of Systems and Software*, 157:110398, 2019. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2019.110398. URL `https://www.sciencedirect.com/science/article/pii/S0164121219301736`.

[27] Amirhossein Deljouyi, Roham Koohestani, Maliheh Izadi, and Andy Zaidman. Leveraging large language models for enhancing the understandability of generated unit tests. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 1449–1461, 2025. doi: 10.1109/ICSE55347.2025.00032.

[28] Pouria Derakhshanfar, Xavier Devroey, and Andy Zaidman. It is not only about control dependent nodes: Basic block coverage for search-based crash reproduction. In *Search-Based Software Engineering: 12th International Symposium, SSBSE 2020, Bari, Italy, October 7–8, 2020, Proceedings*, page 42–57, Berlin, Heidelberg, 2020. Springer-Verlag. ISBN 978-3-030-59761-0. doi: 10.1007/978-3-030-59762-7_4. URL `https://doi.org/10.1007/978-3-030-59762-7_4`.

[29] Barbara DiCicco-Bloom and Benjamin F Crabtree. The qualitative research interview. *Medical Education*, 40(4):314–321, 2006. doi: https://doi.org/10.1111/j.1365-2929.2006.02418.x. URL `https://asmepublications.onlinelibrary.wiley.com/doi/abs/10.1111/j.1365-2929.2006.02418.x`.

[30] Mary T. Dzindolet, Scott A. Peterson, Regina A. Pomranky, Linda G. Pierce, and Hall P. Beck. The role of trust in automation reliance. *International Journal of Human-Computer Studies*, 58(6):697–718, 2003. ISSN 1071-5819. doi: https://doi.org/10.1016/S1071-5819(03)00038-7. URL `https://www.sciencedirect.com/science/article/pii/S1071581903000387`. Trust and Technology.

[31] Fahad Ebrahim and Mike Joy. Semantic similarity search for source code plagiarism detection: An exploratory study. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1*, ITiCSE 2024, page 360–366, New York,

NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706004. doi: 10.1145/3649217.3653622. URL https://doi.org/10.1145/3649217.3653622.

[32] Khalid El Haji, Carolin Brandt, and Andy Zaidman. Using github copilot for test generation in python: An empirical study. In *Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024)*, AST '24, page 45–55, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400705885. doi: 10.1145/3644032.3644443. URL https://doi.org/10.1145/3644032.3644443.

[33] Michael Ellims, James Bridges, and Darrel C Ince. The economics of unit testing. *Empirical Software Engineering*, 11(1):5–31, 2006.

[34] Wenqi Fan, Yujuan Ding, Liangbo Ning, Shijie Wang, Hengyun Li, Dawei Yin, Tat-Seng Chua, and Qing Li. A survey on rag meeting llms: Towards retrieval-augmented large language models. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '24, page 6491–6501, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400704901. doi: 10.1145/3637528.3671470. URL https://doi.org/10.1145/3637528.3671470.

[35] Daniel Fernandes, Ivan Machado, and Rita Maciel. Tempy: Test smell detector for python. In *Proceedings of the XXXVI Brazilian Symposium on Software Engineering*, SBES '22, page 214–219, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450397353. doi: 10.1145/3555228.3555280. URL https://doi.org/10.1145/3555228.3555280.

[36] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, page 416–419, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450304436. doi: 10.1145/2025113.2025179. URL https://doi.org/10.1145/2025113.2025179.

[37] Gordon Fraser and Andrea Arcuri. Evosuite: On the challenges of test case generation in the real world. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 362–369, 2013. doi: 10.1109/ICST.2013.51.

[38] Vahid Garousi and Barış Küçük. Smells in software test code: A survey of knowledge in industry and academia. *Journal of Systems and Software*, 138:52–81, 2018. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2017.12.013. URL https://www.sciencedirect.com/science/article/pii/S0164121217303060.

[39] Jawad Golzar, Shagofah Noor, and Omid Tajik. Convenience sampling. *International Journal of Education and Language Studies*, 1(2):72–77, 2022. ISSN 2767-4851. doi: 10.22034/ijels.2022.162981. URL https://www.ijels.net/article_162981.html.

[40] Giovanni Grano, Simone Scalabrino, Harald C. Gall, and Rocco Oliveto. An empirical investigation on the readability of manual and generated test cases. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, page 348–351, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450357142. doi: 10.1145/3196321.3196363. URL `https://doi.org/10.1145/3196321.3196363`.

[41] Siqi Gu, 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.

[42] Shu Hu. *Pretesting*, pages 5048–5052. Springer Netherlands, Dordrecht, 2014. ISBN 978-94-007-0753-5. doi: 10.1007/978-94-007-0753-5_2256. URL `https://doi.org/10.1007/978-94-007-0753-5_2256`.

[43] Ken Huang, Grace Huang, Adam Dawson, and Daniel Wu. *GenAI Application Level Security*, pages 199–237. Springer Nature Switzerland, Cham, 2024. ISBN 978-3-031-54252-7. doi: 10.1007/978-3-031-54252-7_7. URL `https://doi.org/10.1007/978-3-031-54252-7_7`.

[44] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *ACM Trans. Softw. Eng. Methodol.*, July 2025. ISSN 1049-331X. doi: 10.1145/3747588. URL `https://doi.org/10.1145/3747588`. Just Accepted.

[45] Sašo Karakatič, Aleksej Miloševič, and Tjaša Heričko. Software system comparison with semantic source code embeddings. *Empirical Softw. Engg.*, 27(3), May 2022. ISSN 1382-3256. doi: 10.1007/s10664-022-10122-9. URL `https://doi.org/10.1007/s10664-022-10122-9`.

[46] Jonathan Katzy, Razvan Mihai Popescu, Arie van Deursen, and Maliheh Izadi. The heap: A contamination-free multilingual code dataset for evaluating large language models. In *2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge)*, pages 151–155, 2025. doi: 10.1109/Forge66646.2025.00025.

[47] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. The stack: 3 tb of permissively licensed source code, 2022. URL `https://arxiv.org/abs/2211.15533`.

[48] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc. ISBN 9781713871088.

[49] Hans-Alexander Kruse, Tim Puhlfür, and Walid Maalej. Can developers prompt? a controlled experiment for code documentation generation. In *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 574–586, 2024. doi: 10.1109/ICSME58944.2024.00058.

[50] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 919–931. IEEE, 2023.

[51] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder: may the source be with you!, 2023. URL `https://arxiv.org/abs/2305.06161`.

[52] Jenny T. Liang, Chenyang Yang, and Brad A. Myers. A large-scale survey on the usability of ai programming assistants: Successes and challenges. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400702174. doi: 10.1145/3597503.3608128. URL `https://doi.org/10.1145/3597503.3608128`.

[53] Rensis Likert. A technique for the measurement of attitudes. *Archives of psychology*, 1932.

[54] Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Ruifeng Wang, Zhen Yang, Li Zhang, Zhongqi Li, and Yuchi Ma. Exploring and evaluating hallucinations in llm-powered code generation. *arXiv preprint arXiv:2404.00971*, 2024.

[55] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Comput. Surv.*, 55(9), January 2023. ISSN 0360-0300. doi: 10.1145/3560815. URL `https://doi.org/10.1145/3560815`.

[56] Andrea Lops, Fedelucio Narducci, Azzurra Ragone, Michelantonio Trizio, and Claudio Bartolini. A system for automated unit test generation using large language models and assessment of generated test suites. In *2025 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 29–36, 2025. doi: 10.1109/ICSTW64639.2025.10962454.

[57] Stephan Lukasczyk and Gordon Fraser. Pynguin: Automated unit test generation for python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 168–172, 2022.

[58] Ggaliwango Marvin, Nakayiza Hellen, Daudi Jjingo, and Joyce Nakatumba-Nabende. Prompt engineering in large language models. In I. Jeena Jacob, Selwyn Piramuthu, and Przemyslaw Falkowski-Gilski, editors, *Data Intelligence and Cognitive Informatics*, pages 387–402, Singapore, 2024. Springer Nature Singapore. ISBN 978-981-99-7962-2.

[59] Noble Saji Mathews and Meiyappan Nagappan. Design choices made by llm-based test generators prevent them from finding bugs, 2024.

[60] G. Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley Signature Series (Fowler). Pearson Education, 2007. ISBN 9780132797467. URL `https://books.google.nl/books?id=-izOiCEIABQC`.

[61] Nada Mouman. Online Appendix for "PyTestGuard: An IDE- Integrated Tool for Supporting Developers with LLM-Generated Unit Tests" , August 2025. URL `https://doi.org/10.5281/zenodo.17013158`.

[62] Wendkûuni C. Ouédraogo, Yinghua Li, Abdoul Kader Kaboré, Xunzhu Tang, Anil Koyuncu, Jacques Klein, David Lo, and Tégawendé F. Bissyandé. Test smells in llm-generated unit tests. *ArXiv*, abs/2410.10628, 2024.

[63] Wendkûuni C. Ouédraogo, Kader Kaboré, Yinghua Li, Haoye Tian, Anil Koyuncu, Jacques Klein, David Lo, and Tegawendé F. Bissyandé. Large-scale, independent and comprehensive study of the power of llms for test case generation. *ArXiv*, abs/2407.00225, 2024.

[64] Stack Overflow. Technology — 2024 stack overflow developer survey, 2024. URL `https://survey.stackoverflow.co/2024/technology`.

[65] Carlos Pacheco and Michael D. Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*, OOPSLA '07, page 815–816, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595938657. doi: 10.1145/1297846.1297902. URL `https://doi.org/10.1145/1297846.1297902`.

[66] Fabio Palomba, Annibale Panichella, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. Automatic test case generation: what if test code quality matters? In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 130–141, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450343909. doi: 10.1145/2931037.2931057. URL `https://doi.org/10.1145/2931037.2931057`.

[67] Rangeet Pan, Myeongsoo Kim, Rahul Krishna, Raju Pavuluri, and Saurabh Sinha. Aster: Natural and multi-language unit test generation with llms. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 413–424, 2025. doi: 10.1109/ICSE-SEIP66354.2025.00042.

[68] Sebastiano Panichella, Annibale Panichella, Moritz Beller, Andy Zaidman, and Harald C. Gall. The impact of test case summaries on bug fixing performance: an empirical investigation. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 547–558, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450339001. doi: 10.1145/2884781.2884847. URL `https://doi.org/10.1145/2884781.2884847`.

[69] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. On the distribution of test smells in open source android applications: an exploratory study. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, CASCON '19, page 193–202, USA, 2019. IBM Corp.

[70] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. tsdetect: an open source test smells detection tool. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 1650–1654, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370431. doi: 10.1145/3368089.3417921. URL `https://doi.org/10.1145/3368089.3417921`.

[71] Rafiqul Rabin, Jesse Hostetler, Sean McGregor, Brett Weir, and Nick Judd. Sandboxeval: Towards securing test environment for untrusted code, 2025. URL `https://arxiv.org/abs/2504.00018`.

[72] Chaiyong Ragkhitwetsagul and Jens Krinke. Siamese: scalable and incremental code clone search via multiple code representations. *Empirical Softw. Engg.*, 24 (4):2236–2284, August 2019. ISSN 1382-3256. doi: 10.1007/s10664-019-09697-7. URL `https://doi.org/10.1007/s10664-019-09697-7`.

[73] Sara Rahimi and Marzieh khatooni. Saturation in qualitative research: An evolutionary concept analysis. *International Journal of Nursing Studies Advances*, 6:100174, 2024. ISSN 2666-142X. doi: https://doi.org/10.1016/j.ijnsa.2024.100174. URL `https://www.sciencedirect.com/science/article/pii/S2666142X24000018`.

[74] Martin Riddell, Ansong Ni, and Arman Cohan. Quantifying contamination in evaluating code generation capabilities of language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 14116–14137, 2024.

[75] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2023.

[76] Pranab Sahoo, Ayush Kumar Singh, Sriparna Saha, Vinija Jain, Samrat Mondal, and Aman Chadha. A systematic survey of prompt engineering in large language models: Techniques and applications, 2025. URL `https://arxiv.org/abs/2402.07927`.

[77] Railana Santana, Luana Martins, Larissa Rocha, Tássio Virgínio, Adriana Cruz, Heitor Costa, and Ivan Machado. Raide: a tool for assertion roulette and duplicate assert identification and refactoring. In *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*, SBES '20, page 374–379, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450387538. doi: 10.1145/3422392.3422510. URL `https://doi.org/10.1145/3422392.3422510`.

[78] Arkadii Sapozhnikov, Mitchell Olsthoorn, Annibale Panichella, Vladimir Kovalenko, and Pouria Derakhshanfar. Testspark: Intellij idea's ultimate test generation companion. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, pages 30–34, 2024.

[79] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, 50(1):85–105, 2023.

[80] Ebert Schoofs, Mehrdad Abdi, and Serge Demeyer. Ampyfier: Test amplification in python. *Journal of Software: Evolution and Process*, 34(11):e2490, 2022. doi: https://doi.org/10.1002/smr.2490. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2490`.

[81] Jiho Shin, Clark Tang, Tahmineh Mohati, Maleknaz Nayebi, Song Wang, and Hadi Hemmati. Prompt engineering or fine-tuning: An empirical assessment of llms for code. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*, pages 490–502, 2025. doi: 10.1109/MSR66628.2025.00082.

[82] Mohammed Latif Siddiq, Joanna Cecilia Da Silva Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinícius Carvalho Lopes. Using large language models to generate junit tests: An empirical study. In *Proceedings of the 28th international conference on evaluation and assessment in software engineering*, pages 313–322, 2024.

[83] Auste Simkute, Lev Tankelevitch, Viktor Kewenig, Ava Elizabeth Scott, Abigail Sellen, and Sean Rintel. Ironies of generative ai: Understanding and mitigating productivity loss in human-ai interaction. *International Journal of Human–Computer Interaction*, 41(5):2898–2919, 2025. doi: 10.1080/10447318.2024.2405782. URL `https://doi.org/10.1080/10447318.2024.2405782`.

[84] Davide Spadini, Fabio Palomba, Andy Zaidman, Magiel Bruntink, and Alberto Bacchelli. On the relation of test smells to software code quality. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 1–12, 2018. doi: 10.1109/ICSME.2018.00010.

[85] Florian Tambon, Arghavan Moradi-Dakhel, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Giuliano Antoniol. Bugs in large language models generated code: an empirical study. *Empirical Softw. Engg.*, 30(3), February 2025. ISSN 1382-3256. doi: 10.1007/s10664-025-10614-4. URL `https://doi.org/10.1007/s10664-025-10614-4`.

[86] Yutian Tang, Zhijie Liu, Zhichao Zhou, and Xiapu Luo. Chatgpt vs sbst: A comparative assessment of unit test suite generation. *IEEE Trans. Softw. Eng.*, 50(6): 1340–1359, June 2024. ISSN 0098-5589. doi: 10.1109/TSE.2024.3382365. URL `https://doi.org/10.1109/TSE.2024.3382365`.

[87] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. Unit test case generation with transformers and focal context, 2020.

[88] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*, CHI EA ’22, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391566. doi: 10.1145/3491101.3519665. URL `https://doi.org/10.1145/3491101.3519665`.

[89] Arie Van Deursen, Leon Moonen, Alex Van Den Bergh, and Gerard Kok. Refactoring test code. In *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*, pages 92–95. Citeseer, 2001.

[90] Edwin Van Teijlingen and Vanora Hundley. The importance of pilot studies. *Nursing Standard (through 2013)*, 16(40):33, 2002.

[91] Tássio Virgínio, Luana Martins, Larissa Rocha, Railana Santana, Adriana Cruz, Heitor Costa, and Ivan Machado. Jnose: Java test smell detector. In *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*, SBES ’20, page 564–569, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450387538. doi: 10.1145/3422392.3422499. URL `https://doi.org/10.1145/3422392.3422499`.

[92] Han Wang, Han Hu, Chunyang Chen, and Burak Turhan. Chat-like asserts prediction with the support of large language model. *arXiv preprint arXiv:2407.21429*, 2024.

[93] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*, 50(4):911–936, 2024. doi: 10.1109/TSE.2024.3368208.

[94] Tongjie Wang, Yaroslav Golubev, Oleg Smirnov, Jiawei Li, Timofey Bryksin, and Iftekhar Ahmed. Pynose: a test smell detector for python. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, ASE '21, page 593–605. IEEE Press, 2022. ISBN 9781665403375. doi: 10.1109/ASE51524.2021.9678615. URL https://doi.org/10.1109/ASE51524.2021.9678615.

[95] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models, 2023. URL https://arxiv.org/abs/2203.11171.

[96] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. Codet5+: Open code large language models for code understanding and generation. 2023. URL https://arxiv.org/abs/2305.07922.

[97] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc. ISBN 9781713871088.

[98] Justin D. Weisz, Shraddha Vijay Kumar, Michael Muller, Karen-Ellen Browne, Arielle Goldberg, Katrin Ellice Heintze, and Shagun Bajpai. Examining the use and impact of an ai code assistant on developer productivity and experience in the enterprise. In *Proceedings of the Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*, CHI EA '25, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400713958. doi: 10.1145/3706599.3706670. URL https://doi.org/10.1145/3706599.3706670.

[99] Yuhao Wu, Franziska Roesner, Tadayoshi Kohno, Ning Zhang, and Umar Iqbal. Isolategpt: An execution isolation architecture for llm-based agentic systems, 2025. URL https://arxiv.org/abs/2403.04960.

[100] Chunli Xie, Xia Wang, Cheng Qian, and Mengqi Wang. A source code similarity based on siamese neural network. *Applied Sciences*, 10(21), 2020. ISSN 2076-3417. doi: 10.3390/app10217519. URL https://www.mdpi.com/2076-3417/10/21/7519.

[101] Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, and Junjie Chen. On the evaluation of large language models in unit test generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, page 1607–1619, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400712487. doi: 10.1145/3691620.3695529. URL `https://doi.org/10.1145/3691620.3695529`.

[102] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: deliberate problem solving with large language models. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc.

[103] Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. Evaluating and improving chatgpt for unit test generation. *Proceedings of the ACM on Software Engineering*, 1(FSE):1703–1726, 2024.

[104] Morteza Zakeri-Nasrabadi, Saeed Parsa, Mohammad Ramezani, Chanchal Roy, and Masoud Ekhtiarzadeh. A systematic literature review on source code similarity measurement and clone detection: Techniques, applications, and challenges. *Journal of Systems and Software*, 204:111796, 2023. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2023.111796. URL `https://www.sciencedirect.com/science/article/pii/S0164121223001917`.

[105] Quanjun Zhang, Ye Shang, Chunrong Fang, Siqi Gu, Jianyi Zhou, and Zhenyu Chen. Testbench: Evaluating class-level test case generation capability of large language models. *arXiv preprint arXiv:2409.17561*, 2024.

[106] Ziyao Zhang, Chong Wang, Yanlin Wang, Ensheng Shi, Yuchi Ma, Wanjun Zhong, Jiachi Chen, Mingzhi Mao, and Zibin Zheng. Llm hallucinations in practical code generation: Phenomena, mechanism, and mitigation. *Proc. ACM Softw. Eng.*, 2 (ISSTA), June 2025. doi: 10.1145/3728894. URL `https://doi.org/10.1145/3728894`.

[107] Sebastian Zhao, Alan Zhu, Hussein Mozannar, David Sontag, Ameet Talwalkar, and Valerie Chen. Codinggenie: A proactive llm-powered programming assistant. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, FSE Companion '25, page 1168–1172, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400712760. doi: 10.1145/3696630.3728603. URL `https://doi.org/10.1145/3696630.3728603`.

[108] Sol Zilberman and H. C. Betty Cheng. "no free lunch" when using large language models to verify self-generated programs. In *2024 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 29–36, 2024. doi: 10.1109/ICSTW60967.2024.00018.

Appendix A

Questionnaires

This appendix contains the questionnaires and form used during the user study process. Section A.1 presents the informed consent form, which we sent via email to the participants before the interview. Sections A.2 and A.3 show the pretest and postquestionnaires used during the interview to gather feedback from the participants.

A.1 Informed Consent Form

You are being invited to participate in a research study titled “PyTestGuard: Unit Test Generation Tool for PyCharm”. This study is being done by Nada Mouman from the TU Delft as part of her master’s thesis, under the supervision of Carolin Brandt

The purpose of this research study is to investigate the challenges developers face when generating unit tests for Python code using large language models (LLMs). During the study, you will be introduced to a plugin called PyTestGuard along with a Python project. You will be asked to generate high-quality unit tests using two different versions of PyTestGuard. During the interview, you will be asked some questions and also to fill out a survey to share your opinions on PyTestGuard and on the use of LLMs for generating unit tests in general.

The interview will be conducted over a video call (MS Team) and will take approximately **45-60** minutes to complete. The data collected from the interview and survey will be used for Nada Mouman’s MSc thesis report. In the form below, you will be asked to indicate whether you agree with recording the interview, allowing us to review and analyse it at a later stage. To facilitate this analysis, the recording will be transcribed into a written script, and any information that could identify you will be removed to ensure your anonymity. If you do not agree to the interview being recorded, we can proceed without recording. Survey responses will be collected anonymously. You may choose to opt out of having your anonymized survey data included in the final dataset by indicating your preference in the form below.

As with any online activity the risk of a breach is always possible. To the best of our ability your answers in this study will remain confidential. We will minimize any risks by ensuring the survey is completely anonymous and restricting access to the data only to the research team. The results presented in the thesis report will be completely anonymous and any personal data (name, email address, interview recording, transcript) will be destroyed after completing the study.

Your participation in this study is entirely voluntary and you can withdraw at any time before its completion in September 2025. You are free to omit any questions. The data collected during the interview will be anonymised and any personal information will be destroyed after completing the study.

You have the option to indicate in the form below whether any response or input you provided during the interview may be anonymously quoted in our research outputs. Additionally, you can specify whether your survey data may be included in an aggregated and anonymized dataset. If you indicate 'no', your quotes will be excluded from the thesis report, and your data will not be included in the final dataset.
If you have any additional questions about the interview process or data handling, you can contact the researcher for clarification.

A.1.1 General Agreement – Research Goals, Participant Tasks and Voluntary Participation

1. I have read and understood the study information today, or it has been read to me. I have been able to ask questions about the study and my questions have been answered to my satisfaction.

   [Multiple choice]: Yes, No

2. I consent voluntarily to be a participant in this study and understand that I can refuse to answer questions and I can withdraw from the study at any time before its completion in September 2025, without having to give a reason.

   [Multiple choice]: Yes, No

3. I understand that taking part in the study also involves participating in an interview where I will use PyTestGuard to complete some tasks about unit testing.

   [Multiple choice]: Yes, No

4. I understand that taking part in the study also involves filling out a survey questionnaire about my user experience.

   [Multiple choice]: Yes, No

A.1.2 Potential Risks of Participating (Including Data Protection)

1. I agree to the recording of the interview video call, including audio, video, and any shared screens. These recordings will be securely stored and destroyed after the completion of the study. All results that may be included in reports or publications will be anonymized.

   [Multiple choice]: Yes, No

2. I understand that the textualized script of the interview will be collected for further analysis. These transcripts will be securely stored and destroyed after the completion of the study. All personal and identifying information (such as my name) will be removed to protect my identity. All results that may be included in reports or publications will be anonymized.

   [Multiple choice]: Yes, No

3. I understand that I will have the opportunity to review the transcript of my interview to ensure it is accurate and to request that specific statements be amended or removed.

   [Multiple choice]: Yes, No

4. I understand that my responses to the accompanying survey are collected anonymously and will not be linked to my personal identity.

   [Multiple choice]: Yes, No

5. I understand that personal information collected about me that can identify me, such as name and email, video recordings and transcripts, will not be shared beyond the study team.

   [Multiple choice]: Yes, No

6. I understand that the following steps will be taken to minimize the threat of a data breach and protect my identity in the event of such a breach:

   - My data will be stored securely, with anonymized data uploaded to a secure TU Delft OneDrive.
   - Techniques such as anonymization and data aggregation will be used to protect my identity and prevent re-identification.

   [Multiple choice]: Yes, No

7. I understand that if I choose to withdraw from the study, all data I have provided, including interview recordings, transcripts and survey responses, will be permanently deleted and will not be included in any study results.

   [Multiple choice]: Yes, No

8. I understand that the (identifiable) personal data I provide will be destroyed after the study is complete (September 2025).

   [Multiple choice]: Yes, No

A.1.3 Research Publication, Dissemination and Application

1. I understand that after the research study the de-identified information I provide in the interview and will be used for Nada Mouman's thesis report.

   [Multiple choice]: Yes, No

2. I understand that the anonymous survey responses I provide will be aggregated with those of other participants and used for Nada Mouman's thesis report.

   [Multiple choice]: Yes, No

3. I agree that the responses, views or other input that I give during the interview can be quoted anonymously in research outputs.

   [Multiple choice]: Yes, No

4. I agree that the anonymous survey responses I provide can be aggregated and used in research outputs.

   [Multiple choice]: Yes, No

5. I give permission for the anonymized quotes that I provide to be archived in 4TU.ResearchData so it can be used for future research and learning.

   [Multiple choice]: Yes, No

6. I give permission for the aggregated data that I provide to be archived in 4TU.ResearchData so it can be used for future research and learning.

   [Multiple choice]: Yes, No

A.1.4 Acknowledgment of Consent

By proceeding with this study, you acknowledge that you have read and understood this information and agree to participate.

1. I sign this consent agreement with the following name:

   [Open]

2. I sign this consent agreement on the following day:

   [Open]

A.2 Pretest Questionnaire

Welcome to the PyTestGuard research study survey! Thank you for participating in this study exploring unit test generation for Python using Large Language Models. Your feedback will help us better understand developer needs and improve tools like PyTestGuard. The open-ended questions will be discussed during the interview, so you don't need to fill them out here.

A.2.1 Demographics

1. How many years of programming experience do you have?

   [Multiple choice]: 0-1 years, 2-5 years, 6-10 years, 10+ years

2. Which programming languages do you mainly use?

   [Open]

3. What is your professional role?

   [Multiple choice]: Software developer, Student, Researcher, Other

4. How often do you write unit tests?

   [Multiple choice]: Never, Occasionally, Frequently, Always

5. What challenges do you usually face when writing unit tests?

   [Open]

A.2.2 Experience with Large Language Models

1. Have you used any automated test generation tools (e.g., Pynguin, Randoop, EvoSuite, TestSpark)? If yes, which ones?

   [Multiple choice]: Yes, No

2. How likely are you to use Large Language Models (LLMs) to generate unit tests in your development workflow?

   [Likert 1-5]

3. How would you rate your experience with LLMs for generating unit tests?

   [Likert 1-5] with N/A option

4. What features do you expect from a tool that helps generate unit tests using LLMs?

   [Open]

A.3 Posttest Questionnaire

A.3.1 First Task: Evaluation of the PyTestGuard v1

Generate unit tests for a given method using the first version of PyTestguard, selecting unittest as the test framework. Then, you can rewrite and fix the generated tests to ensure they work correctly.

1. I trust that the unit tests generated by PyTestGuard are understandable and mostly correct.

   [Likert 1-5]

2. I am confident that the final tests I wrote with PyTestGuard are runnable, meaningful, and of good quality.

   [Likert 1-5]

3. What challenges did you face when writing unit tests while using this version of PyTestGuard?

   [Open]

A.3.2 Second Task: Evaluation of the PyTestGuard v2

Generate unit tests for a given method using the second version of PyTestguard, selecting unittest as the test framework. Then, rewrite and fix the generated tests to ensure they work correctly. Make full use of the features provided by this version (e.g., highlights, running tests).

1. I trust that the unit tests generated by PyTestGuard are understandable and mostly correct.

   [Likert 1-5]

2. I am confident that the final tests I wrote with PyTestGuard are runnable, meaningful, and of good quality.

   [Likert 1-5]

3. What challenges did you face when writing unit tests while using this version of PyTestGuard?

   [Open]

A.3.3 Comparison between the two version of PyTestGuard

1. Which version of the tool did you prefer overall?

   [Multiple choice]: Simple, Enhanced, No Preference

2. Which version gave you more confidence in the quality of the generated tests?

   [Multiple choice]: Simple, Enhanced, No Preference

A.3.4 User Experience with Enhanced PyTestGuard

The next questions will ask about your experience using the enhanced version of PyTest-Guard.

1. PyTestGuard helped me identify and fix issues in the generated tests (e.g., syntax errors, test smells, hallucinations).

   [Likert 1-5]

2. The information provided (e.g., execution result, coverage, test smells) was clear and useful.

   [Likert 1-5]

3. How PyTestGuard helped you spot or fix issues in generated tests (e.g., syntax errors, test smells, hallucinations)?

   [Open]

4. Which of the following PyTestGuard features did you find most useful? Please rank them by importance.

   [Options]: Test smell detection, Summarized execution error message, Execution results (green/red), Statement Coverage, Statement Coverage Change, Inspections for issues in LLM-generated tests

5. Do you think additional features are needed? If yes, what would be most helpful?

   [Open]

A.3.5 Assessment of PyTestGuard as a Testing Assistant

1. After using PyTestGuard, you are more likely to use Large Language Models (LLMs) to generate unit tests in my development workflow.

   [Likert 1-5]

2. PyTestGuard matched your expectations for an LLM-based unit test assistant.

   [Likert 1-5]

3. You would want to use PyTestGuard to help you write unit tests in the future.

   [Likert 1-5]

4. It makes sense for PyTestGuard to be integrated into an IDE (e.g. PyCharm).

   [Likert 1-5]

5. How does it compare to other tools you've used for generating unit tests (e.g., ChatGPT, GitHub Copilot)?

   [Open]

6. Do you have any other remarks or feedback you would like to share?

   [Open]

Appendix B

Functions Used in the User Study

This appendix contains the Python function used to complete the tasks during the user study.

B.1 booking.py

```python
"""Algorithms for 'booking' inventory, that is, the process of finding a
matching lot when reducing the content of an inventory.
"""

__copyright__ = "Copyright (C) 2015-2018, 2020-2021, 2024  Martin Blais"
__license__ = "GNU GPLv2"

from beancount.core import amount
from beancount.core import position
from beancount.core.number import MISSING
from beancount.core.position import CostSpec

def convert_spec_to_cost(units, cost_spec: CostSpec | None):
    """Convert a posting's CostSpec instance to a Cost.

    Args:
      units: An instance of Amount.
      cost_spec: An instance of CostSpec.
    Returns:
      An instance of Cost.
    """
    cost = cost_spec
    if isinstance(units, amount.Amount):
        if cost_spec is not None:
            number_per, number_total, cost_currency, date, label, merge =
 cost_spec
            units_num = units.number

            # Compute the cost.
            if (number_per is not MISSING or number_total is not None)
 and not units.number.is_zero():
                if number_total is not None:
                    # Compute the per-unit cost if there is some total cost
                    # component involved.
                    cost_total = number_total
                    if number_per is not MISSING:
                        cost_total += number_per * units_num
                    unit_cost = cost_total / abs(units_num)
                else:
                    unit_cost = number_per
                cost = position.Cost(unit_cost, cost_currency, date, label)
            else:
                cost = None
    return cost
```

B.2 realization.py

```python
__copyright__ = "Copyright (C) 2013-2021, 2024  Martin Blais"
__license__ = "GNU GPLv2"

import collections

from beancount.core import account
from beancount.core.data import Balance
from beancount.core.data import Close
from beancount.core.data import Custom
from beancount.core.data import Document
from beancount.core.data import Note
from beancount.core.data import Open
from beancount.core.data import Pad
from beancount.core.data import Transaction
from beancount.core.data import TxnPosting

def postings_by_account(entries):
    """Create lists of postings and balances by account.

    This routine aggregates postings and entries grouping them by account name.
    The resulting lists contain postings in-lieu of Transaction directives, but
    the other directives are stored as entries. This yields a list of postings
    or other entries by account. All references to accounts are taken into
    account.

    Args:
      entries: A list of directives.
    Returns:
       A mapping of account name to list of TxnPosting instances or
       non-Transaction directives, sorted in the same order as the entries.
    """
    txn_postings_map = collections.defaultdict(list)
    for entry in entries:
        if isinstance(entry, Transaction):
            # Insert an entry for each of the postings.
            for posting in entry.postings:
                txn_postings_map[posting.account].append(TxnPosting(entry
    , posting))

        elif isinstance(entry, (Open, Close, Balance, Note, Document)):
            # Append some other entries in the realized list.
            txn_postings_map[entry.account].append(entry)

        elif isinstance(entry, Pad):
            # Insert the pad entry in both realized accounts.
            txn_postings_map[entry.account].append(entry)
            txn_postings_map[entry.source_account].append(entry)

        elif isinstance(entry, Custom):
            # Insert custom entry for each account in its values.
            for custom_value in entry.values:
                if custom_value.dtype == account.TYPE:
                    txn_postings_map[custom_value.value].append(entry)

    return txn_postings_map
```