22APCS, FIT, HCMUS - VNUHCM

AI Software Testing Tools

Group009: Seminar Report

Written by : Nguyễn Bạch Trường Giang - 2212502
Nguyễn Vĩnh Khang - 22125035
Đặng Minh Nhựt - 22125071
Lưu Quốc Bảo - 22125008

Dec 2025

Contents

1 Introduction

Software testing plays an important role in the Software Development Life Cycle. It ensures the software operates as expected. Despite the undeniable contribution of software testing to the software's quality, it takes much of a project's recources [1].

Test automation was created to reduce the time and effort spent on testing. In particular, testers write scripts to automatically execute repetitive tasks for the System Under Test (SUT). It has been studied that automated testing brings postive impacts to the product [2]. However, the scripts need to be maintained after code changes, and the time-consuming process prior to test execution is still of humans' responsibility.

AI for software testing is another approach to support testers with their work. Indeed, AI-based testing solutions have been witnessing a rise in popularity during the past decade [3]. Acordding to a survey in 2017 [4], 2025 was the year that second most people predicted for manual and automation testing replacement with AI.

Conincidentally, this report is written at the very end of 2025. We would love to discuss about the current state of AI software testing tools, particularly in test case design and unit testing. In addition, we will conduct small experiments to compare the performance of AI tools. As a side quest, we can answer for 2017 survey-takers whether AI has yet dominated the testing world.

2 Test Case Design

2.1 The life cycle of software testing

A tester follows the Software Testing Life Cycle (STLC) with 6 stages (see Figure 11):

1. Requirements analysis: analyze and clarify the requirements documentation/user stories to understand what need to be tested
2. Test planning: decide what to do in this testing phase and what not based on the available resources and the situation at the moment
3. Test design: convert general requirements into test cases and detail what need to be in the test data as well as the necessary tools
4. Environment setup: prepare the tools and environments (e.g. virtual machines, emulators)
5. Test execution: execute the test cases with the test data constructed in step 3; report the results in some form of documentation (e.g. tracking systems, Excel sheet)
6. Test closure: finalize the test project with lessons, potential improvements, archived resources and reports of the overall process

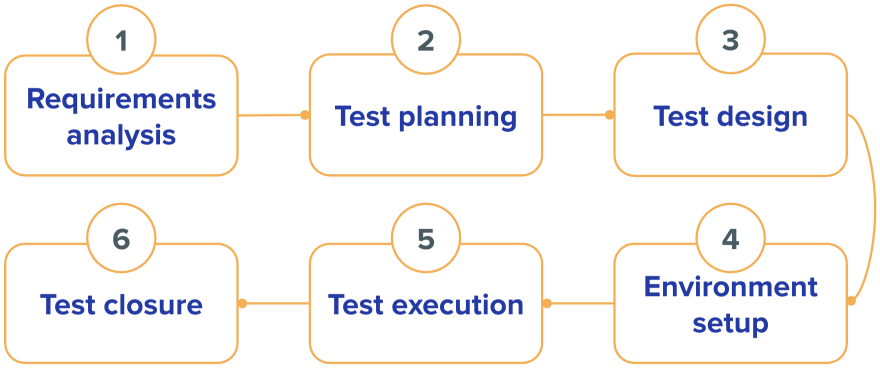

Figure 1: Software Testing Life Cycle

2.2 How can AI support the flow?

The test execution, which is resource-consuming by nature when done manually, has been aided by automation tools for some time. Considering the remaining stages, test design requires much of the time and effort.

There are endless possibilities for test cases, and testers must carefully consider what to use based on the characteristics of the products. Although there are several

methods to narrow down the amount of test cases such as domain testing, scenario testing, or state-transition testing, the remaining is still of a great number.

Also, admist the tediousness of designing test cases, humans can make errors even without them noticing due to several reasons, including fatigues and stress. It is even more likely for errors to happen again after some first errors [5].

To alleviate the burden created by prolonged tasks, AI, can be leveraged. Particularly, LLMs support testers with test case and test data generation.

2.2.1 Document-like context

One type of input for models is requirements specification or user stories. The models subsequently generate several test cases and the set of test data designed based on the prompts. Testers, at this moment, have more time for strategic thinking.

General-purpose LLMs such as ChatGPT, Gemini or Claude can be easily accessed by anyone, but it comes with the disadvantage of serving too broad a domain. Testers must polish the prompts carefully and provide enough context, even including code and internal materials, to use these tools effectively. Otherwise, these models will only touch the "tip of the iceberg".

AI-powered testing tools, on the other hand, use algorithms specifically target issues in the testing industry. This reduces the need for a highly refined prompt, but not every individual can get the license for the tools.

2.2.2 Product-like context

Another input type is the GUI. In this case, a model handles both scenario and test case design. This is more suitable for systems whose documentations are not available such as the toolshop website. Also, GUI-based test cases are closer to real-life user experience and more explicit to reproduce the bug.

This solution shares similarities with exploratory or ad-hoc testing, requiring freedom in thinking and human-like actions. The difference is instead of manually "destroying" the SUT, we let machines perform the task. There is a saying that highlights the role of these testing techniques: "There are known knowns. There are things we know that we know. There are known unknowns. That is to say, there are things that we now know we don't know. But there are also unknown unknowns. There are things we do not know we don't know.", said Donald Rumsfeld in 2008. Exploring software without pre-defined constraints leaves space for the discovery of those "unknown unknowns". On the other hand, documentations are to ensure the "known knowns", and in some cases "known unknowns".

An example for such a tool is Sapienz [6], a Meta's project. It is designed to work with different sources of inputs: source code (white-box) or APK (can, grey-box, or cannot be reverse-engineered, black-box). Then, it extracts *motif genes* combining various *atomic genes*. An atomic gene can be an elementary action performed on the GUI such as touching, rotating or pinching and zooming. The motif genes are based on the UI of the AUT in the current view. Atomic genes add randomness to the test case while motif genes are systematic. This hybrid usage ensures a fuller overall coverage. Then, these test cases are fed into a cycle where they are evaluated with a fitness function that satisfies multiple objectives including coverage level, sequence

length and number of crashes. This fitness score is used as a base for the genetic algorithm. This cycle is repeated and forms a search-based software testing solution.

2.3 How much does AI support the flow?

According to an article of AWS [7], using AI for test case generation in automotive testing has reduced up to 80% of the time required normally. This proves that in industry, AI can boost the performance of testers.

3 Test Case Design - Experiment

In this section, we evaluate the performance of popular LLMs on Test case design. We choose Gemini 3 Pro and GPT-5.1. We feed them with 3 level of Prompting: (1) User Story + Acceptance Criteria (AC), (2) User Story + AC + Testing Technique, and (3) User Story + AC + Testing Technique + High-level Test Case. The metric for evaluation is Bug Coverage.

3.1 Experiment Setup

For each LLM, we create a new chat for each input and turn on the "Thinking" mode. We also turn off the "learn from past chat" feature of Gemini since we don't want the current response to have reference on earlier responses.

After receiving 6 test suites from LLMs, we automate the test execution step using Pytest and Selenium. We implemented the scripts using the design pattern Page Object Model (POM). The code mainly consists: a script for interaction with the feature's web page, 6 scripts for each test suite, and configurations scripts. A screenshot of the code is shown in Figure 2.

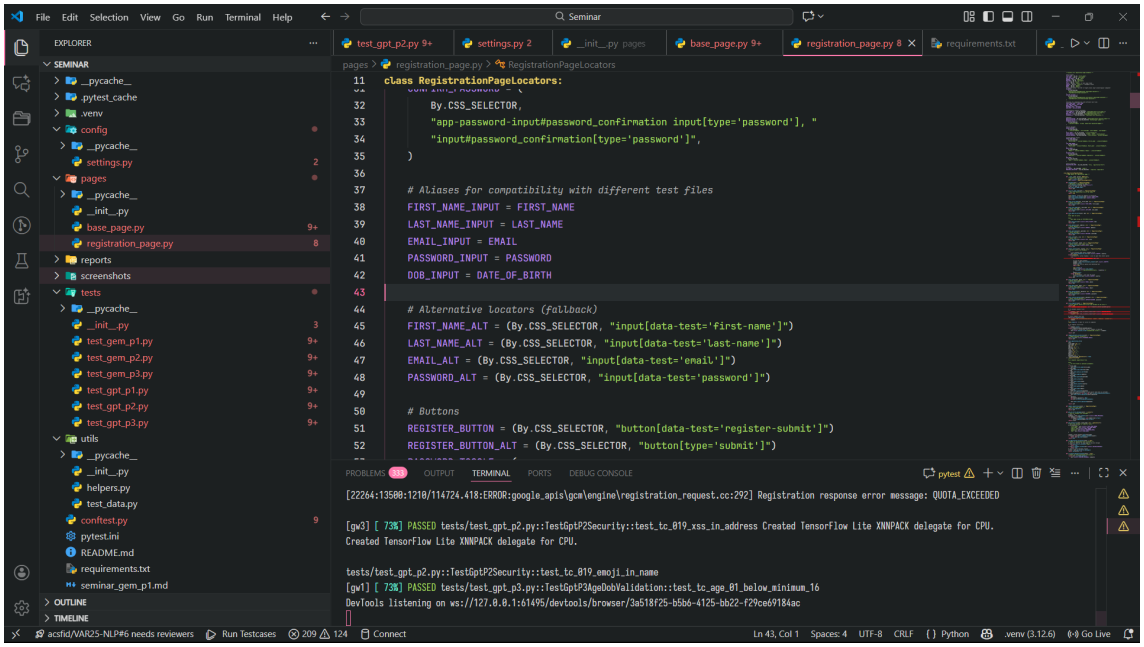

Figure 2: Automation test scripts to execute generated test suites.

This is a part of the test script for Gemini + Prompt 1 generated test suite:

```
...
    @pytest.mark.security
    @pytest.mark.high
    @pytest.mark.parametrize(
        "password,missing_requirement",
        [
            ("password123!", "uppercase"),
            ("PASSWORD123!", "lowercase"),
```

```
            ("PasswordOne!", "digit"),
            ("Password123", "symbol/special character"),
        ],
    )
    def test_sec_02_password_complexity_content(
        self,
        registration_page: RegistrationPage,
        password: str,
        missing_requirement: str,
    ):
        """
        TC: SEC-02 - Password Complexity: Content

        Test passwords missing specific complexity requirements.

        Expected: Error message explaining specific missing complexity
            requirement.
        """
        user_data = TestDataGenerator.get_valid_user()

        registration_page.fill_registration_form(
            first_name=user_data.first_name,
            last_name=user_data.last_name,
            dob=user_data.dob,
            address=user_data.address,
            postcode=user_data.postcode,
            city=user_data.city,
            state=user_data.state,
            country=user_data.country,
            phone=user_data.phone,
            email=user_data.email,
            password=password,
            confirm_password=password,
        )

        registration_page.click_register()

        errors = registration_page.get_all_error_messages()

        assert (
            len(errors) > 0
        ), f"Should show error for password missing {missing_requirement}.
            Password: {password}"
...
```

The contents of 3 levels of prompt are listed follow:
Prompt 1

```
## Role
You are a Senior Software Quality Assurance (QA) Engineer with over 10 years
    of experience in Manual Testing. You have a "destructive mindset" and
    excel at identifying edge cases, logic flaws, and potential security
    vulnerabilities that developers often overlook.

## Objective
```

```
Your task is to analyze a provided Software Specification (or User Story) along with a screenshot of the feature's UI, then design a comprehensive Manual Test Suite along with a robust set of Test Data that Tester can use to test the feature. The Test Cases are in table format ready to be used.

## Specification
User Story:
As a visitor, I want to register an account so that I can log in and use personalized features.

Acceptance Criteria:
- Age must be between 17 and 75 inclusive.
- Email must be unique and in valid format.
- All mandatory fields (first name, last name, email, password, confirm password) must be filled.
- Name/address/phone fields respect max lengths and validation rules.
- Password must be at least 8 characters, contain upper, lower, digit, and symbol, not be compromised, and match confirm password.

Steps to access register form:
1. Access: [Practice Software Testing - Toolshop - v5.0 with bugs](https://with-bugs.practicesoftwaretesting.com/)
2. Click on "Sign in" button on the top right corner
3. Click on "Register your account" in the Login form

Existed Account:
- First name: Jane
- Last name: Doe
- E-mail: customer@practicesoftwaretesting.com
- Password: welcome01
```

Prompt 2

```
## Role
You are a Senior Software Quality Assurance (QA) Engineer with over 10 years of experience in Manual Testing. You have a "destructive mindset" and excel at identifying edge cases, logic flaws, and potential security vulnerabilities that developers often overlook.

## Objective
Your task is to analyze a provided Software Specification (or User Story) along with a screenshot of the feature's UI, then design a comprehensive Manual Test Suite along with a robust set of Test Data that Tester can use to test the feature. The Test Cases are in table format ready to be used.

## Specification
User Story:
As a visitor, I want to register an account so that I can log in and use personalized features.

Acceptance Criteria:
- Age must be between 17 and 75 inclusive.
- Email must be unique and in valid format.
```

```
- All mandatory fields (first name, last name, email, password, confirm password) must be filled.
- Name/address/phone fields respect max lengths and validation rules.
- Password must be at least 8 characters, contain upper, lower, digit, and symbol, not be compromised, and match confirm password.

Steps to access register form:
1. Access: [Practice Software Testing - Toolshop - v5.0 with bugs](https://with-bugs.practicesoftwaretesting.com/)
2. Click on "Sign in" button on the top right corner
3. Click on "Register your account" in the Login form

Existed Account:
- First name: Jane
- Last name: Doe
- E-mail: customer@practicesoftwaretesting.com
- Password: welcome01

Testing Techniques to Apply
- BVA -> for age and password length (bugs usually hide at edges).
- EP -> for valid/invalid email, password character rules, mandatory fields filled/missing.
- Decision Table -> for combinations of missing fields.
- Error Guessing -> try SQL injection, emoji, spaces in fields.
```

Prompt 3

```
## Role
You are a Senior Software Quality Assurance (QA) Engineer with over 10 years of experience in Manual Testing. You have a "destructive mindset" and excel at identifying edge cases, logic flaws, and potential security vulnerabilities that developers often overlook.

## Objective
Your task is to analyze a provided Software Specification (or User Story) along with a screenshot of the feature's UI, then design a comprehensive Manual Test Suite along with a robust set of Test Data that Tester can use to test the feature. The Test Cases are in table format ready to be used.

## Specification
User Story:
As a visitor, I want to register an account so that I can log in and use personalized features.

Acceptance Criteria:
- Age must be between 17 and 75 inclusive.
- Email must be unique and in valid format.
- All mandatory fields (first name, last name, email, password, confirm password) must be filled.
- Name/address/phone fields respect max lengths and validation rules.
- Password must be at least 8 characters, contain upper, lower, digit, and symbol, not be compromised, and match confirm password.

Steps to access register form:
```

```
1. Access: [Practice Software Testing - Toolshop - v5.0 with bugs](https://
    with-bugs.practicesoftwaretesting.com/)
2. Click on "Sign in" button on the top right corner
3. Click on "Register your account" in the Login form

Existed Account:
- First name: Jane
- Last name: Doe
- E-mail: customer@practicesoftwaretesting.com
- Password: welcome01

Testing Techniques to Apply
- BVA -> for age and password length (bugs usually hide at edges).
- EP -> for valid/invalid email, password character rules, mandatory fields
    filled/missing.
- Decision Table -> for combinations of missing fields.
- Error Guessing -> try SQL injection, emoji, spaces in fields.

High-Level Test Cases
- Age: Test 16, 17, 75, 76 (boundary values).
- Email: Test valid email, invalid formats, already-used email.
- Mandatory Fields: Leave one field empty, then multiple fields, confirm
    messages appear correctly.
- Max Lengths: Test exactly at max length, and one character over.
- Password Strength: Try missing each character class, too short, and a
    compromised password (if system supports it).
```

3.2 Results

The execution process yields 34 failed, 165 passed, 1 skipped, 15 xfailed in 1998.56s (0:33:18) overall. Total unique bugs detected is xxxx. To evaluate the quality of the test suites generated, we use Bug Coverage, which is calculated by:

$$\text{Bug Coverage} = \frac{\text{Detected Bug by Test Suite}}{\text{All Bugs of the Features}} \times 100\%$$

We don't take LLMs' response time into account since we believe that the human review time for the responses will be much more than the LLM performance. The statistics of the test suites' size and bug coverage evaluation are illustrated in Table 1 and 2.

The result shows that Gemini generated much less test cases than GPT while having similar bug coverage. Moreover, only GPT with Prompt 2 detect the "Email with double dots not rejected" bug. The limitation of this experiment is that we don't take into account of the randomness when generate of the LLMs. For more reliable results, we must generate more test suites per (LLM, Prompt) tuple.

	GPT-5.1			Gemini 3 Pro		
	#TC	**BugCov**	**BugCov/#TC**	**#TC**	**BugCov**	**BugCov/#TC**
Prompt 1	32	75.00%	234.38%	15	75.00%	500.00%
Prompt 2	25	75.00%	300.00%	17	75.00%	441.18%
Prompt 3	54	87.50%	162.04%	17	75.00%	441.18%

Table 1: Performance Metrics per Prompt

Bugs	**GPT-P1**	**GPT-P2**	**GPT-P3**	**Gem-P1**	**Gem-P2**	**Gem-P3**
Age 17 not allowed	x	x	x	x	x	x
Age >75 not rejected	x	x	x	x	x	x
DOB Leap day allow	x		x		x	
pwd miss uppercase	x	x	x	x	x	x
pwd miss lowercase	x		x	x		x
pwd miss digit	x	x	x	x	x	x
pwd miss symbol		x	x	x	x	x
email w double dots		x				

Table 2: Bug Detection Matrix

4 Unit Testing

4.1 Unit Testing and The Code Coverage Metric

In the domain of software engineering, unit testing is defined as the practice of verifying the smallest testable parts of an application, typically functions or methods, in isolation from the rest of the system. The primary goal is to validate that each component performs exactly as designed before it is integrated into the larger architecture.

To measure the effectiveness of unit testing, the industry relies heavily on Code Coverage—a metric indicating the percentage of source code executed by the test suite. Theoretically, higher coverage correlates with a lower probability of undetected bugs. However, achieving high coverage manually is notoriously labor-intensive. Developers often face the "diminishing returns" curve: writing tests for the main logic (the "happy path") is straightforward, but covering the remaining 10-20

This is where the theoretical limitations of manual testing meet the capabilities of Artificial Intelligence. The promise of AI in unit testing is not merely to write code faster, but to systematically explore the "search space" of possible inputs that humans often overlook due to cognitive bias or fatigue.

4.2 The flow of unit testing

Similar to the Software Testing Life Cycle (STLC), the process of unit testing follows a structured workflow, often integrated directly into the development environment (IDE) and Continuous Integration (CI) pipelines:

Component Analysis: The developer analyzes the function or method to understand its logic, input parameters, expected return values, and potential exceptions.

Test Case Design: Designing specific scenarios, including "happy paths" (standard execution) and "edge cases" (boundary conditions like empty lists, null values, or negative numbers).

Implementation: Writing the actual test code using a testing framework (e.g., JUnit for Java, Pytest for Python). This involves setting up the test data (Arrangement), executing the function (Act), and verifying the result (Assert).

Execution: Running the tests locally or via a CI server.

Refactoring and Maintenance: If a test fails, the code is fixed. If the code structure changes, the tests must be updated to reflect the new reality.

This cycle is repeated continuously. The philosophy of Test-Driven Development (TDD) even suggests writing the test before the actual code, though in practice, many developers write tests concurrently or immediately after implementation.

4.3 Theoretical Approaches to AI-Driven Unit Testing

AI does not apply a single monolithic approach to unit testing. From a theoretical standpoint, current AI solutions leverage two distinct algorithmic paradigms: Search-Based Software Testing (SBST) and Large Language Models (LLMs).

4.3.1 Search-Based Software Testing (SBST)

This approach, grounded in optimization theory, treats test generation as a search problem. Tools using this method (such as those based on genetic algorithms) do not "read" code in the human sense. Instead, they generate a population of random test cases and evolve them using a fitness function.

The Mechanism: The fitness function typically prioritizes code coverage. The algorithm executes random inputs against the software. If a specific input manages to execute a previously unvisited branch of code (e.g., a complex if-else condition), that test case is "selected" and mutated further.

Theoretical Value: SBST excels at finding "corner cases"—mathematically precise inputs that trigger specific boundary conditions (e.g., integer overflows or null pointer exceptions) that are counter-intuitive to human logic.

4.3.2 LLM-Based Generation (Neural Approaches)

In contrast, the more recent wave of AI (Generative AI) views unit testing as a Language Translation task. Models like GPT-4 or Codex are trained on vast repositories of code (e.g., GitHub). They learn the statistical probability of which test code follows a given function signature.

The Mechanism: When a developer provides a function, the LLM predicts the most statistically probable test inputs and assertions based on patterns it has seen in millions of other projects.

Theoretical Value: Unlike SBST, LLMs understand "semantics" and naming conventions. They can infer the intent of a function from its name (e.g., calcu-

lateInvoiceTotal) and generate readable, human-like test code that serves as excellent documentation.

4.4 The "Test Oracle" Problem and The Coverage Illusion

While AI automation is powerful, it introduces significant theoretical risks that must be managed. The most prominent challenge is known as the "Test Oracle Problem".

A "Test Oracle" is the mechanism that determines whether a test has passed or failed. In manual testing, the developer is the oracle: they know that for input A, the output should be B.

- The Challenge: AI tools can easily generate input A (the setup) and execute the code (the action). However, determining the correct output B (the assertion) is difficult because the AI does not truly "understand" the business requirements. It only sees the code.
- Regression vs. Correctness: Most AI tools assume the current behavior of the code is correct. If the code contains a bug, the AI may generate a test that asserts the buggy result is "correct." This makes AI excellent for Regression Testing (ensuring behavior doesn't change) but dangerous for Correctness Testing (ensuring behavior is right).

This leads to the "Coverage Illusion". It is possible for an AI to generate a test suite that achieves 100% code coverage—executing every single line of the application—without actually testing the logic. For instance, a generated test might call a function but lack meaningful assertions. The metrics look perfect, but the safety net is non-existent.

4.5 The Paradigm Shift: From Author to Auditor

The integration of AI into unit testing necessitates a fundamental shift in the developer's role. We are moving away from the era of the "Test Author"—who types every line of boilerplate code—to the era of the "Test Auditor" or "Reviewer."

In this new theoretical model, the AI acts as a "stochastic generator," producing a high volume of potential test scenarios and boilerplate scaffolding. The human developer's responsibility shifts to:

- Verification of Intent: Ensuring the AI-generated assertions align with the actual business logic, not just the code's current behavior.
- Constraint Management: Guiding the AI to avoid testing implementation details that make tests brittle.
- Ethical Oversight: Ensuring that data fed into AI models for test generation does not leak proprietary algorithms or sensitive user data.

In conclusion, AI in unit testing is not a "silver bullet" that eliminates human effort. Instead, it changes the nature of the effort. It solves the problem of "writer's block" and coverage gaps, but it elevates the need for deep code comprehension. As articulated in recent software engineering studies, the value of AI lies in its ability

to handle the repetition of testing, allowing humans to focus on the correctness of the system

5 Unit Testing - Experiment

5.1 Tools Introduction

5.1.1 UnitTestAI

UnitTestAI determines the programming language of the chosen code block based on the current file extension. The addon gives the user a list of relevant test frameworks to select from once the language has been identified. This implies that UnitTestAI can be used by developers with any preferred test framework and programming language. They only need to pick the right framework, let the extension identify the language, and choose the code they wish to test. Because of this capability, UnitTestAI is incredibly flexible and easy to use, enabling developers to utilize the tools and workflows of their choice without having to learn a new framework or language-specific testing methods.

Figure 3: UnitTestAI

5.1.2 Qodo

Qodo is an AI-powered tool that provides Local Code Review directly within the Integrated Development Environment (IDE), aiming to shift code reviews left and enhance developer confidence. It functions by using custom-defined Agents, which operate in continuous Modes (like Ask, Code, or Plan) or single-task Workflows (for tasks like test generation or documentation), both of which are shareable for standardization. Qodo analyzes code changes with multi-repo context and organization-specific rules to deliver precise, high-signal feedback, detect bugs and standard violations as you code, and offers features like 1-click issue resolution and test generation for code changes, ultimately helping developers push cleaner code and improve overall code quality.

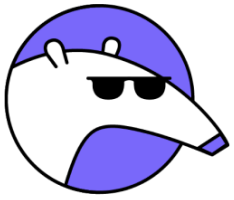

Figure 4: Qodo

5.2 Experiment Design

This experiment is designed to evaluate the effectiveness of two distinct AI-powered code analysis and testing tools, UnitTestAI and Qodo, in the context of generating high-quality test cases for Python code. The core process involves utilizing a common dataset consisting of 10 test cases sourced from the Hugging Face platform, ensuring a standardized benchmark for comparison. Each tool will be independently applied to generate test suites for the code snippets within this dataset. The generated test cases from both UnitTestAI and Qodo will then be rigorously assessed and compared against three primary criteria: Code Coverage, which measures the amount of source code executed by the tests; Test Suite Efficiency, which likely evaluates the speed, non-redundancy, and execution time of the generated tests; and Readability and Documentation, which assesses how clear, understandable, and well-documented the generated test code is. The goal is to determine which tool offers superior performance across these three dimensions, providing insight into their respective strengths and weaknesses for automating the test-generation phase of the software development lifecycle.

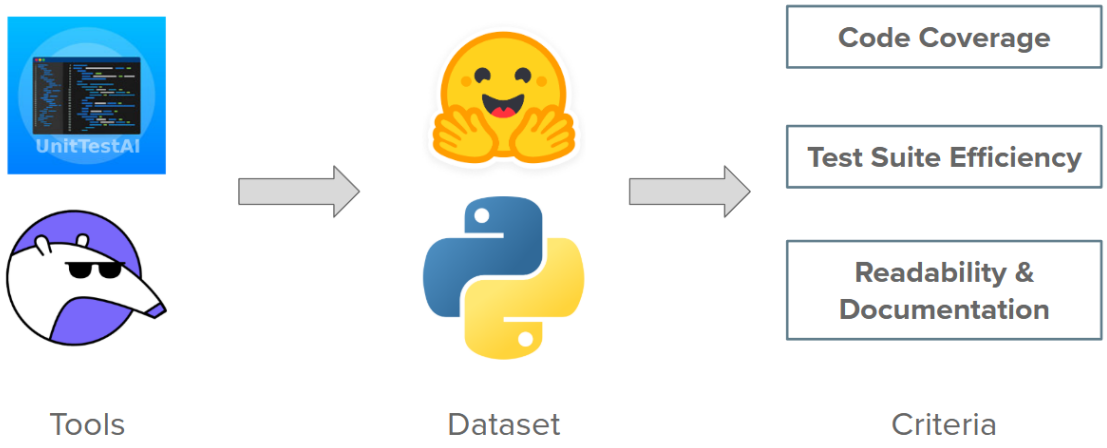

Figure 5: Experiment Plan

5.3 Execution

5.3.1 UnitTestAI

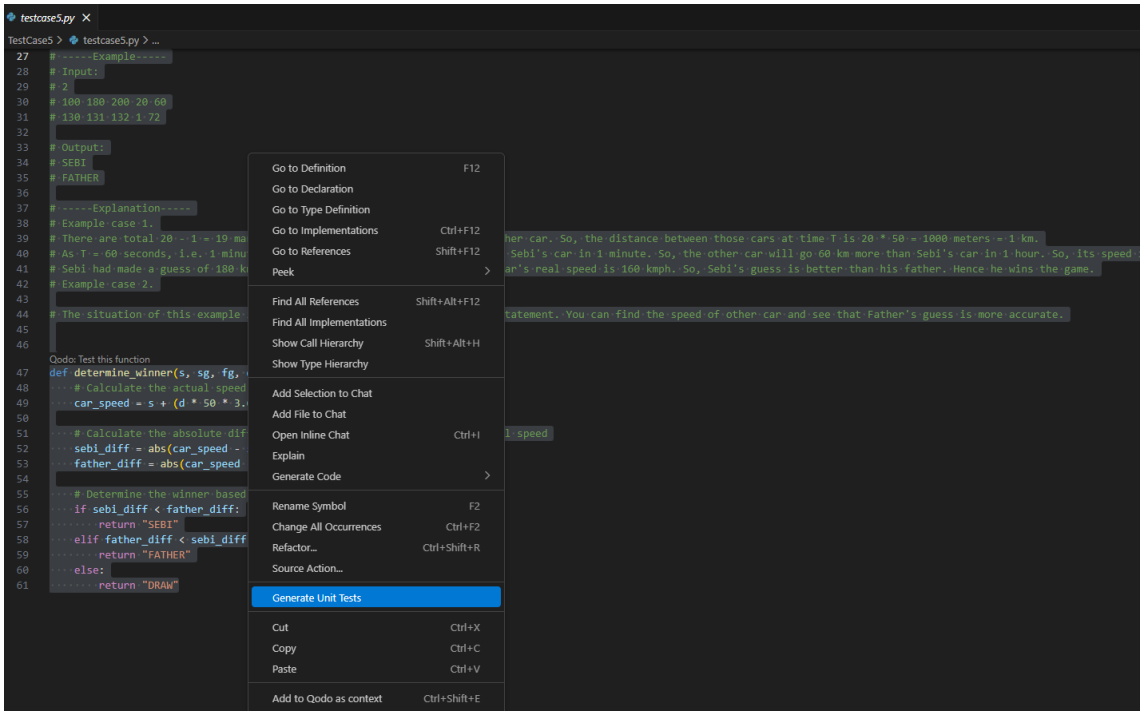

Figure 6: Step 1: Select the code as well as the context. Then click "Generate Unit Tests".

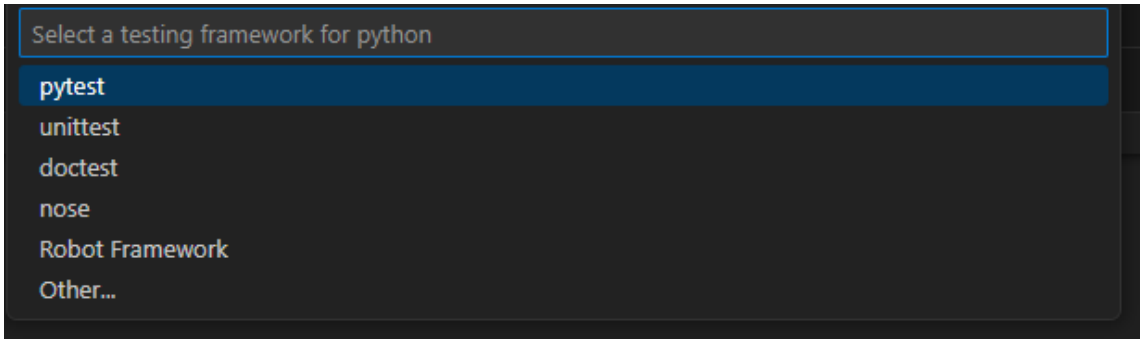

Figure 7: Step 2: The tool will analyze the programming language and let you choose your testing framework.

```python
//python
import unittest

Qodo: Test this function
def determine_winner(s, sg, fg, d, t):
    # Calculate the actual speed of the other car
    car_speed = s + (d * 50 * 3.6) / t

    # Calculate the absolute differences between the guesses and the actual speed
    sebi_diff = abs(car_speed - sg)
    father_diff = abs(car_speed - fg)

    # Determine the winner based on the differences
    if sebi_diff < father_diff:
        return "SEBI"
    elif father_diff < sebi_diff:
        return "FATHER"
    else:
        return "DRAW"


class TestSpeedGuessGame(unittest.TestCase):

    def test_positive_basic_cases(self):
        self.assertEqual(determine_winner(100, 180, 200, 20, 60), "SEBI")
        self.assertEqual(determine_winner(130, 131, 132, 1, 72), "FATHER")

    def test_draw_case(self):
        self.assertEqual(determine_winner(100, 145, 145, 10, 60), "DRAW")

    def test_high_speed_and_close_guess(self):
        self.assertEqual(determine_winner(200, 300, 301, 50, 60), "SEBI")

    def test_low_speed_and_far_guess(self):
        self.assertEqual(determine_winner(50, 100, 200, 10, 60), "FATHER")

    def test_edge_case_minimum_input(self):
        self.assertEqual(determine_winner(0, 0, 0, 0, 1), "DRAW")

    def test_edge_case_high_difference(self):
        self.assertEqual(determine_winner(0, 1000, 1500, 100, 1), "SEBI")

    def test_edge_case_huge_distance_and_short_time(self):
        self.assertEqual(determine_winner(0, 2000, 3000, 1000, 1), "FATHER")
```

Figure 8: Step 3: Wait for the test cases to be generated and execute them.

5.3.2 Qodo

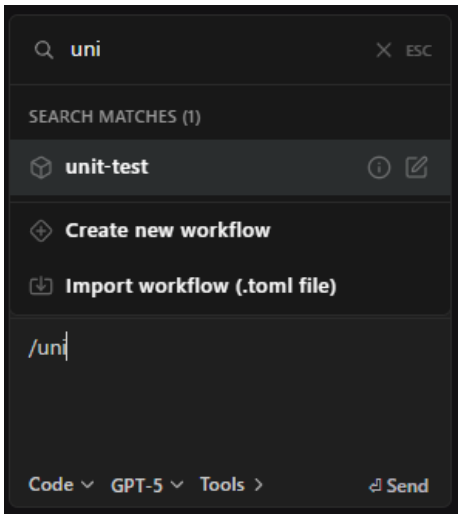

Figure 9: Step 1: Choose specific action for unit test.

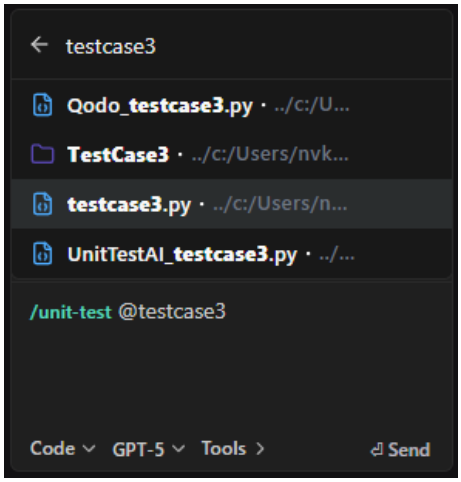

Figure 10: Step 2: Include context files

Include

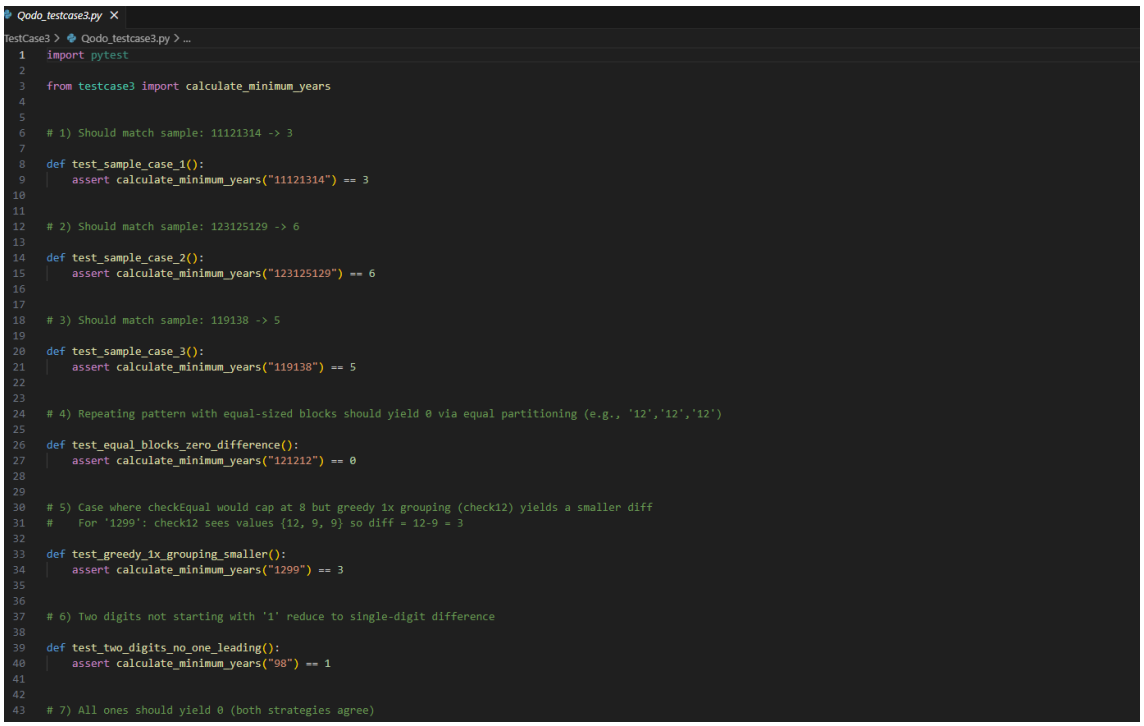

Figure 11: Step 3: Wait for the test cases to be generated and execute them.

5.4 Experiment result

5.4.1 Example

Testcase 1:

There are n candy boxes in front of Tania. The boxes are arranged in a row from left to right, numbered from 1 to n. The i-th box contains ri candies, candies have the color c_i (the color can take one of three values — red, green, or blue). All candies inside a single box have the same color (and it is equal to ci).

Initially, Tanya is next to the box number s. Tanya can move to the neighbor box (that is, with a number that differs by one) or eat candies in the current box. Tanya eats candies instantly, but the movement takes one second.

If Tanya eats candies from the box, then the box itself remains in place, but there is no more candies in it. In other words, Tanya always eats all the candies from the box and candies in the boxes are not refilled.

It is known that Tanya cannot eat candies of the same color one after another (that is, the colors of candies in two consecutive boxes from which she eats candies are always different). In addition, Tanya's appetite is constantly growing, so in each next box from which she eats candies, there should be strictly more candies than in the previous one.

Note that for the first box from which Tanya will eat candies, there are no restrictions on the color and number of candies.

Tanya wants to eat at least k candies. What is the minimum number of seconds she will need? Remember that she eats candies instantly, and time is spent only on movements.

—Input—

The first line contains three integers n, s and k ($1 \leq n \leq 50$, $1 \leq s \leq n$, $1 \leq k \leq 2000$) — number of the boxes, initial position of Tanya and lower bound on number of candies to eat. The following line contains n integers r_i ($1 \leq r_i \leq 50$) — numbers of candies in the boxes. The third line contains sequence of n letters 'R', 'G' and 'B', meaning the colors of candies in the correspondent boxes ('R' for red, 'G' for green, 'B' for blue). Recall that each box contains candies of only one color. The third line contains no spaces.

—Output—

Print minimal number of seconds to eat at least k candies. If solution doesn't exist, print "-1".

—Examples—

Input 5 3 10 1 2 3 4 5 RGBRR

Output 4

Input 2 1 15 5 6 RG

Output -1

—Note—

The sequence of actions of Tanya for the first example:

move from the box 3 to the box 2; eat candies from the box 2; move from the box 2 to the box 3; eat candy from the box 3; move from the box 3 to the box 4; move from the box 4 to the box 5; eat candies from the box 5.

Since Tanya eats candy instantly, the required time is four seconds.

```
def min_seconds_to_eat_candies(n, s, k, r, c):
    INF = 10000000000.0
    max_n = 50
    max_k = 2000

    s -= 1 # Convert to 0-based index
```

```
    buf = [''] * (max_n + 1)
    dp = [[0 for _ in range(max_n + 1)] for _ in range(max_k + 1)]
    answer = INF

    for i in range(len(c)):
        buf[i] = c[i]

    for i in range(k, -1, -1):
        for j in range(n):
            dp[i][j] = INF

    for j in range(n):
        value = abs(j - s)
        if k - r[j] <= 0:
            answer = min(answer, value)
        else:
            dp[k - r[j]][j] = value

    for i in range(k, 0, -1):
        for j in range(n):
            if dp[i][j] < INF:
                for l in range(n):
                    if buf[j] != buf[l] and r[j] < r[l]:
                        value = dp[i][j] + abs(j - l)
                    if i - r[l] <= 0:
                        answer = min(answer, value)
                    else:
                        dp[i - r[l]][l] = min(dp[i - r[l]][l], value)

    return -1 if answer == INF else answer
```

UnitTestAI - Output:

```
import pytest
from unittest.mock import patch
from testcase1 import min_seconds_to_eat_candies

def test_min_seconds_to_eat_candies_positive_simple():
    n = 3
    s = 2
    k = 5
    r = [1, 2, 3]
    c = ['A', 'B', 'A']
    result = min_seconds_to_eat_candies(n, s, k, r, c)
    assert result == 1

def test_min_seconds_to_eat_candies_negative_k_not_enough():
    n = 3
    s = 2
    k = 1
    r = [5, 5, 5]
    c = ['A', 'B', 'A']
    result = min_seconds_to_eat_candies(n, s, k, r, c)
    assert result == -1

def test_min_seconds_to_eat_candies_edge_case_one_candy():
    n = 1
    s = 1
    k = 10
    r = [1]
    c = ['A']
    result = min_seconds_to_eat_candies(n, s, k, r, c)
    assert result == 0

def test_min_seconds_to_eat_candies_at_start_position():
    n = 5
    s = 1
    k = 10
    r = [1, 2, 3, 4, 5]
    c = ['B', 'B', 'A', 'B', 'A']
    result = min_seconds_to_eat_candies(n, s, k, r, c)
    assert result == 0

def test_min_seconds_to_eat_candies_multiple_options():
    n = 4
    s = 3
    k = 10
    r = [1, 1, 1, 1]
    c = ['A', 'A', 'B', 'B']
    result = min_seconds_to_eat_candies(n, s, k, r, c)
    assert result == 1

def test_min_seconds_to_eat_candies_negative_no_possible_moves():
    n = 3
    s = 2
    k = 3
    r = [5, 5, 5]
    c = ['A', 'A', 'A']
    result = min_seconds_to_eat_candies(n, s, k, r, c)
    assert result == -1

def test_min_seconds_to_eat_candies_n_zero():
    n = 0
    s = 1
    k = 5
    r = []
    c = []
```

```
    with pytest.raises(IndexError):
        result = min_seconds_to_eat_candies(n, s, k, r, c)

def test_min_seconds_to_eat_candies_r_less_than_values():
    n = 4
    s = 2
    k = 5
    r = [1, 2, 2]  # insufficient values
    c = ['A', 'B', 'A', 'C']
    with pytest.raises(IndexError):
        result = min_seconds_to_eat_candies(n, s, k, r, c)

def test_min_seconds_to_eat_candies_c_less_than_values():
    n = 4
    s = 3
    k = 5
    r = [1, 2, 3, 1]
    c = ['A', 'B']  # insufficient values
    with pytest.raises(IndexError):
        result = min_seconds_to_eat_candies(n, s, k, r, c)
```

Qodo - Output:

```
import pytest

from testcase1 import min_seconds_to_eat_candies


# 1) Should match the provided sample case #1
# Input:
# 5 3 10
# 1 2 3 4 5
# RGBRR
# Expected minimal seconds: 4

def test_example_case_1():
    n, s, k = 5, 3, 10
    r = [1, 2, 3, 4, 5]
    c = list("RGBRR")
    assert min_seconds_to_eat_candies(n, s, k, r, c) == 4


# 2) Should match the provided sample case #2 (impossible)
# Input:
# 2 1 15
# 5 6
# RG
# Expected: -1

def test_example_case_2_impossible():
    n, s, k = 2, 1, 15
    r = [5, 6]
    c = list("RG")
    assert min_seconds_to_eat_candies(n, s, k, r, c) == -1


# 3) Should return 0 seconds when the starting box alone satisfies k
# Start at box 2 which has 5 candies, k=5.

def test_zero_seconds_when_start_box_suffices():
    n, s, k = 3, 2, 5
    r = [1, 5, 3]
    c = list("RGB")
    assert min_seconds_to_eat_candies(n, s, k, r, c) == 0


# 4) Should be impossible if two consecutive boxes of the same color are required to reach k
# With all boxes the same color and the largest single box < k, you cannot eat from two boxes consecutively.

def test_impossible_due_to_same_color_requirement():
    n, s, k = 3, 2, 4
    r = [2, 2, 3]
    c = list("GGG")
    assert min_seconds_to_eat_candies(n, s, k, r, c) == -1


# 5) Should enforce strictly increasing candies between eaten boxes (equal values not allowed)
# Here, two boxes both with 2 candies would not be a valid sequence to reach k=4.

def test_strictly_increasing_required_equal_not_allowed():
    n, s, k = 3, 2, 4
    r = [2, 2, 2]
    c = list("RGB")  # colors allow alternation, but equal counts disallow sequence
    assert min_seconds_to_eat_candies(n, s, k, r, c) == -1


# 6) Should find a minimal path using increasing candy counts and alternating colors
# Start at box 3 (r=3, color R), then move to box 4 (r=4, color G) to reach k=6 in 1 second total movement.

def test_min_seconds_increasing_and_alternating_path():
    n, s, k = 5, 3, 6
    r = [1, 2, 3, 4, 5]
    c = list("RGRGR")
    assert min_seconds_to_eat_candies(n, s, k, r, c) == 1
```

```
# 7) Should handle edge start position and compute minimal movement correctly
# Start at the first box, need to reach k=7. Best path: box 1 (1, R) -> box 2 (3, G) -> box 4 (5, G?)
# Adjust colors so alternation is enforced. Expect minimal movement calculation to be correct.

def test_minimal_movement_from_edge():
    n, s, k = 5, 1, 7
    r = [1, 3, 4, 5, 2]
    # Colors arranged to allow valid sequences with alternation and strictly increasing candies:
    c = list("RGRGR")
    # One optimal sequence: start at box1 (1, R), then box3 (4, R) is invalid due to same color; instead
    # go box1 (1, R) -> box2 (3, G) [+1 sec], total=4, then box4 (5, G) invalid due to same color; thus pick box3 (4, R) [+1 sec],
    # but 1 -> 3 -> 4 violates strictly increasing from 3 to 4 is OK and colors alternate G->R.
    # However, smarter is to start at box2 (move +1), eat 3 (G), then go to box4 (5, G) invalid; instead to box3 (4, R) [+1],
    # 3 -> 4 is valid and colors alternate; total movement from start: 1 (to 2) + 1 (to 3) = 2.
    # This reaches 7 with 2 seconds. Assert 2.
    assert min_seconds_to_eat_candies(n, s, k, r, c) == 2
```

5.4.2 Summarize

Table 3: Experiment Result

Criterion	Qodo	UnitTest AI
Code Coverage	**Stronger.** Explicitly targeted all possible *logical outcomes* for conditional branches (e.g., Problem 5: SEBI, FATHER, DRAW) and domain-specific failure modes (e.g., Problem 1: impossible color alternation).	**Good.** Covered sample cases and basic boundary conditions (e.g., $N = 1$, impossible K), but often used less realistic/-generic inputs (e.g., 'A', 'B' colors in Problem 1).
Readability and Documentation	**Superior Clarity.** Each test function is often accompanied by detailed comments explaining the *logical purpose*, *expected path*, and *reasoning* for the case, significantly aiding human understanding and debugging.	**Functional Clarity.** Uses clear, descriptive function names, but generally lacks the rich, problem-specific commentary provided by Qodo, relying on the test's input/output to convey its purpose.
Test Structure and Modularity	**High Modularity.** Tends to use dedicated, separate functions for each distinct test case or logical path (e.g., separate `test_sebi_closer`, `test_father_closer`). This makes tests highly focused and easily self-contained.	**High Conciseness.** Favors the use of the `pytest.mark.parametrize` decorator to group similar test inputs, which reduces boilerplate code and makes the overall test file shorter and cleaner.
Test Suite Efficiency	**Sacrifices Speed for Strength.** Introduced heavy-duty helper functions, such as the brute-force optimizer in Problem 2. This approach guarantees the optimal solution is checked, but it can lead to **significantly longer execution times** as the constraints increase.	**Prioritizes Speed.** All tests run quickly as they rely purely on the tested algorithm without computationally expensive verification (like brute force). This makes for faster developer feedback cycles.

6 Conclusion

With all of the above said, it can be seen that AI upgrades the game of testers.

However, the use of AI faces ethical challenges. Data privacy is one of the first things businesses consider. This somehow constraints the power of generative AI models where context matters. OpenAI CEO Sam Altman, in a podcast on Jun 18, stated: "Now that the computer knows a lot of context on me, and if I ask it a question with only a small number of words, it knows enough about the rest of my life to be pretty confident in what I want it to do. Sometimes in ways I don't even think of. I think we are heading towards a world where, if you want, the AI will just have unbelievable context on your life and give you super, super helpful answers." Although the paragraph stemmed from his excitement for the new memory feature of ChatGPT, it poses a threat on business secrets and customers' privacy.

In addition, generative AI is non-deterministic by nature. If these models are let alone with the QA process, the final product is also non-deterministic consequently. This leads to a paradox where QA cannot guarantee the quality of the deliverables.

Therefore, humans play the pivotal role in the testing process. Being over-reliant on AI can reduce the reliability of the software. This stresses the need for a firm knowledge foundation from testers to judge what can be accepted from AI and what to add to the results. In other words, humans need to control AI technologies [8]. After all, the end users are human beings, not machines.

And we have completed the side quest: AI has yet to dominate our world.

References

[1] Norbert Oster and Francesca Saglietti. Automatic test data generation by multi-objective optimisation. In Janusz Górski, editor, *Computer Safety, Reliability, and Security*, pages 426–438, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-45763-3.

[2] Divya Kumar and K.K. Mishra. The impacts of test automation on software's cost, quality and time to market. *Procedia Computer Science*, 79:8–15, 2016. ISSN 1877-0509. doi: https://doi.org/10.1016/j.procs.2016.03.003. URL https://www.sciencedirect.com/science/article/pii/S1877050916001277. Proceedings of International Conference on Communication, Computing and Virtualization (ICCCV) 2016.

[3] Alex Escalante-Viteri and David Mauricio. Artificial intelligence in software testing: A systematic review of a decade of evolution and taxonomy. *Algorithms*, 18(11), 2025. ISSN 1999-4893. URL https://www.mdpi.com/1999-4893/18/11/717.

[4] Tariq M. King, Jason Arbon, Dionny Santiago, David Adamo, Wendy Chin, and Ram Shanmugam. Ai for testing today and tomorrow: Industry perspectives. In *2019 IEEE International Conference On Artificial Intelligence Testing (AITest)*, pages 81–88, 2019. doi: 10.1109/AITest.2019.000-3.

[5] Tyler Adkins, Han Zhang, and Taraz Lee. People are more error-prone after committing an error. *Nature Communications*, 15, 07 2024. doi: 10.1038/s41467-024-50547-y.

[6] Ke Mao, Mark Harman, and Yue Jia. Sapienz: multi-objective automated testing for android applications. In *ISSTA 2016: Proceedings of the 25th International Symposium on Software Testing and Analysis*, page 94–105, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450343909. doi: 10.1145/2931037.2931054. URL https://doi.org/10.1145/2931037.2931054.

[7] Tobias Drees, Daniel Krumpholz, and Stanislav Kruglov. Using generative ai to create test cases for software requirements | aws for industries, Jan 2025. URL https://aws.amazon.com/blogs/industries/using-generative-ai-to-create-test-cases-for-software-requirements/.

[8] Per Erik Strandberg, Eduard Paul Enoiu, and Mirgita Frasheri. Ethical challenges and software test automation. *AI and Ethics*, 5(6):6185–6206, 2025. doi: 10.1007/s43681-025-00804-7. URL https://doi.org/10.1007/s43681-025-00804-7.