How Students Use Generative AI for Software Testing: An Observational Study

Baris Ardic[1*], Quentin Le Dilavrec[1] and Andy Zaidman[1]

[1*]Computer Science, Delft University of Technology, Delft, 2628XE, State, Netherlands.

*Corresponding author(s). E-mail(s): b.ardic@tudelft.nl;
Contributing authors: q.ledilavrec@tudelft.nl; a.e.zaidman@tudelft.nl;

Abstract

The integration of generative AI tools like ChatGPT into software engineering workflows opens up new opportunities to boost productivity in tasks such as unit test engineering. However, these AI-assisted workflows can also significantly alter the developer's role, raising concerns about control, output quality, and learning, particularly for novice developers. This study investigates how novice software developers with foundational knowledge in software testing interact with generative AI for engineering unit tests. Our goal is to examine the strategies they use, how heavily they rely on generative AI, and the benefits and challenges they perceive when using generative AI-assisted approaches for test engineering. We conducted an observational study involving 12 undergraduate students who worked with generative AI for unit testing tasks, using ChatGPT running the GPT-3.5 model. We identified four interaction strategies, defined by whether the test idea or the test implementation originated from generative AI or from the participant. Additionally, we singled out prompting styles that focused on one-shot or iterative test generation, which often aligned with the broader interaction strategy. Students reported benefits including time-saving, reduced cognitive load, and support for test ideation, but also noted drawbacks such as diminished trust, test quality concerns, and lack of ownership. While strategy and prompting styles influenced workflow dynamics, they did not significantly affect test effectiveness or test code quality as measured by mutation score or test smells.

Keywords: software testing, generative AI, human-AI interaction, AI4SE

1 Introduction

The advances of generative artificial intelligence (generative AI) tools are changing how software is developed (Hassan et al., 2024; Russo et al., 2024). While these AI-assisted workflows enable higher productivity (Weber et al., 2024; Ziegler et al., 2024), they likely also result in humans being or feeling less directly involved in how software is written (Schecter and Richardson, 2025). We observe that generative AI becomes increasingly integrated into software engineering workflows, including software testing, where large language models have already been investigated for potential applications to tasks such as unit test generation (Deljouyi et al., 2025; El Haji et al., 2024), test oracle creation (Molina et al., 2025), debugging (Majdoub and Ben Charrada, 2024), and program repair (Wang et al., 2024).

Because of the increasing importance of generative AI-assisted software engineering workflows, it is both important and interesting to better understand how developers interact with these tools in practice. This understanding is fundamental because the effectiveness and reliability of AI-assisted workflows depend not only on the technical capabilities of the tools, but also on how developers use, adapt to, and critically assess their outputs. This is also echoed in the realm of testing, where Ardic et al. have previously called upon the research community to further qualitatively explore how AI tools are used (Ardic et al., 2025a).

Motivated by this call and the growing relevance of human–AI collaboration in software testing, we set out to explore how students, already equipped with foundational knowledge from a dedicated course on software testing (Aniche et al., 2019), make use of generative AI, specifically ChatGPT, when faced with a unit testing task. Our objective was not to assess their theoretical understanding of testing concepts, but to investigate how they intuitively incorporate AI into their workflow.

While our broader motivation is to understand how developers engage with generative AI-assisted testing workflows, we focus in this study on novice testers. Students with foundational knowledge in software testing represent an important segment of emerging AI users: they are both motivated to seek assistance and still in the process of forming testing habits. This makes them a suitable population for examining early-stage interaction patterns with generative AI tools.

By analyzing their prompting strategies, workflows, and reflections, we aim to uncover the skills that are necessary, and the challenges that arise when AI is integrated into test engineering. In doing so, we contribute to the ongoing research agenda on understanding human–AI interaction in software engineering in general (Choudhuri et al., 2024a), and software testing contexts in particular (Ardic et al., 2025a).

The following research questions guide our investigation.

RQ1: What strategies do students adopt when incorporating generative AI into unit testing workflows?

An AI-assisted workflow offers both advantages and potential challenges, making it crucial to understand how students approach working with generative AI in a unit test engineering context. Since there is no predefined user guide or a finite set of best practices for using AI in this context, students must independently develop their

workflows. By investigating the strategies they adopt, we can identify both effective practices and common missteps. This insight allows us to encourage behaviors that maximize the benefits of AI while raising awareness of potential risks.

RQ2: How do students prompt generative AI?

Examining how students formulate their prompts provides valuable insight into how they use generative AI for unit testing.

For example, this can help us uncover the specific use cases they pursued, whether they used generative AI to generate complete test cases, refine the cases they implemented themselves, or improve their understanding of testing concepts.

Analyzing these use patterns helps us understand how students engage with AI in practical testing scenarios and which aspects of AI integration they find most useful.

RQ3: What are the benefits and challenges of a generative AI-assisted test workflow for students?

AI tools promise to improve productivity, but it is essential to evaluate whether students actually experience tangible benefits in terms of efficiency, learning, and software quality. Do they find AI-generated test cases useful? Does AI assistance help them better understand unit testing principles, or does it make them overly reliant on automation? To answer these questions, we examine both the perceived benefits reported by students and the benefits observed by us, researchers. While students can provide insight into how AI affects their workflow, confidence, and learning experience, researchers can analyze whether AI genuinely enhances the quality of their test cases and improves their approach to unit testing.

To address these research questions, we designed an observational study with 12 students who engaged in two consecutive unit testing tasks, during which they were free to use ChatGPT as they saw fit. Our goal was not to assess their theoretical knowledge of testing concepts, since all participants had previously completed a foundational course on software testing, but to observe how they integrated AI into their workflows.

Our study makes four main contributions:

- We identify the test engineering strategies students employ when using generative AI and categorize them into four distinct types.
- We analyze the prompting behaviors of students across assignments, uncovering patterns in how they formulate requests and structure their generative AI collaboration.
- We examine both the perceived and observed benefits and drawbacks of using generative AI for software testing, including impacts on workflow, trust, and learning.
- We propose a conceptual framework that synthesizes our empirical observations into actionable mitigation strategies for the drawbacks of using generative AI,

offering guidance on how to use these tools safely and productively in software testing.

The remainder of the paper is structured as follows. Section 2 outlines our methodology, including experimental setup, data collection, and data analysis process. Section 3 presents our results, organized around the three research questions. Section 4 discusses key insights and implications. Section 5 reflects on threats to validity. Section 6 reviews related work on generative AI in software engineering, and Section 7 concludes the paper.

2 Methodology

We conducted an observational study to examine how students interact with generative AI during unit testing, capturing both behavioral data and participant reflections through screen recordings, think-aloud protocols, and post-task interviews. Section 2.1 describes the experimental setup, tasks, and interview design. Section 2.2 elaborates on our reasoning for employing qualitative methods. The remaining subsections describe our data analysis procedures, organized by research questions.

2.1 Experimental Setup

At the start of the experimental session, each participant completed a short questionnaire to provide relevant background information. Subsequently, they proceeded with two consecutive unit testing tasks, during which they were free to use ChatGPT in any way they found helpful. The participants used the free browser-based version of ChatGPT, which at the time was running GPT-3.5. The experiment was conducted over a period spanning from December 2023 to May 2024.

Throughout the session, we encouraged participants to think out loud, enabling us to capture their reasoning and decision-making processes in real time. In parallel, we recorded observation notes to document the behavior and interactions of the participants as they worked. We also recorded their screen activity, including all prompts and outputs, to enable detailed reconstruction of their workflows. Once participants indicated that they had completed both tasks, we conducted a post-task interview to reflect on their experiences and explore the strategies they employed. This mixed-source dataset comprising screen recordings, prompt logs, observation notes, and interviews enables a fine-grained analysis of their interactions with generative AI. The overall structure of the experimental session used for data collection is summarized in Figure 1.

The unit testing tasks used in this experiment were originally used in a study by Aniche et al. (2022), which investigated how developers engineer test cases; their study was done without the possibility of making use of generative AI. Reusing these assignments enables us to make some comparisons between our findings and theirs, particularly in terms of code metrics related to the resulting test suites.

The two tasks were designed around two methods that were taken from Apache Commons Lang by Apache Software Foundation (2024)[1]. The two methods were chosen such that they do not depend on any other class, and they are relatively short (between 20 and 30 lines of code) string manipulation methods. The Java classes used in the experiment, which contain the implementations of these methods, can be found in Appendix A. For each task, participants were asked to write unit tests for one of the methods that were devoid of any existing unit tests. The two methods are as follows:

- initials(String str, char... delimiters): Extracts the initial characters from each word in the String. All first characters after the defined delimiters are returned as a new string.
- substringsBetween(String str, String open, String close): Searches a String for substrings delimited by a start and end tag, returning all matching substrings in an array.

In addition, we adopt a think-aloud approach similar to that used in their study, where participants are asked to verbalize what they are doing and why. In our instructions to the participants, we advise them to try and answer these questions to help them think out loud:

- What are you doing right now?
- Why would you test this?
- What challenges are you facing?
- What are you not understanding?
- What is the next step?

[1]https://commons.apache.org/lang hash for the commit used: e0b474c0d015f89a52.

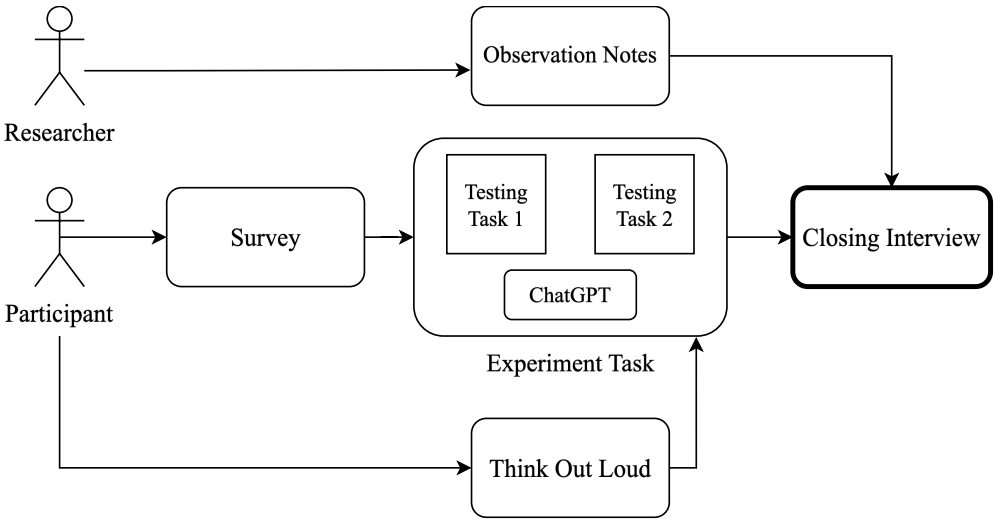

Fig. 1: Experiment Flow

Specifically, while interacting with ChatGPT, the following questions were relevant to think out loud:

- Why are you writing this prompt?
- What do you think of the generated answer?

The think-aloud approach proved useful in helping us capture and structure observation notes, and also enhanced the value of the video recordings by providing a narrative of the participant's actions.

We conducted three pilot sessions to refine our experimental design and rehearse data collection procedures. These sessions informed the development of our think-aloud questions and contributed to the formulation of the interview guidelines.

To better understand participant behaviour, we visualized our observation notes as flow diagrams. Figure 2 shows an example with two flow diagrams from the same participant, illustrating how their approach evolved between assignments. In these diagrams, each block represents an observed participant action during the task (e.g., reviewing the original method documentation provided in the assignment, asking GPT for tests, or adding generated tests to the test suite). Because the diagrams reflect observed behaviour rather than a predefined workflow, the specific actions and their ordering may differ between assignments. In this example, we can observe a change in the strategy from the first assignment (left-hand side) to the second (right-hand side). In particular, the second workflow includes more interactions with GPT for generating and evaluating tests, whereas the first workflow contains more actions related to manual inspection of the method and manual test creation. Both assignments started with the participant asking GPT to generate tests and continued with evaluating and adding generated tests to their test suites, alongside manually creating tests. In the second assignment, the participant put GPT in a more central position in their workflow and created fewer test cases themselves. These diagrams help make the progression and variation in participants' strategies more transparent and easier to detect.

We recruited students from TU Delft on the condition that they had passed the first-year mandatory course on software testing and quality (Aniche et al., 2019; Ardic and Zaidman, 2023). Among twelve participants, eight were in their second year of the bachelor's programme, three were in their third year, and one was a master's student; the bachelor's programme lasts three years. Recruitment was conducted via an internal university Mattermost (2024) channel. The sessions were carried out in a 1 on 1 setting where the researchers took observation notes that were later enhanced by the screen recordings of the participants.

Prior work shows that when interacting with new technologies, students and professionals tend to perform similarly, with experience differences emerging primarily when familiar techniques are applied (Salman et al., 2015). Empirical SE guidelines also emphasize that students are appropriate participants when they match the population of interest (Falessi et al., 2018). Because novice testers are among the groups most likely to seek AI assistance in unit testing, students with foundational software testing knowledge form an appropriate population for our purposes.

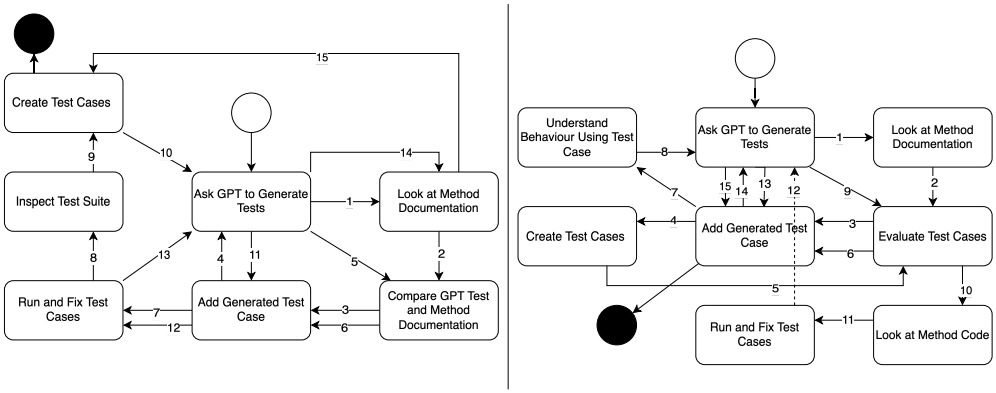

Fig. 2: Flow-diagram examples from each experiment task from a pilot study participant. States represent participant actions, and state transitions are numbered chronologically to indicate the sequence of steps taken.

This study was reviewed and approved by the Human Research Ethics Committee of Delft University of Technology. Participants gave informed consent to take part in the study and were made aware that all findings would be published in anonymized form, with no personally identifiable information disclosed.

2.1.1 Interview Guide

To complement our observational data, we conducted post-task interviews with participants. These interviews were semi-structured, allowing us to explore core themes while also following up on interesting or unexpected behaviors observed during the task. In particular, we expanded our questioning when a participant demonstrated a unique approach or mindset that had not been seen up to that point in the data collection process.

The following questions formed the basis of our interview guide:

- Do you think you could do this task entirely by yourself without needing any help?
- And what has being able to use ChatGPT changed for you?
- Do you think working the way you just did changes which skills you use while you are working?
- Do you think working the way you did requires a different mindset?
- What do you think about the test suite you now have?
- Do you think you can improve it in any way?

2.2 Use of Qualitative Methods

Our study aims to understand how students interact with generative AI during unit testing tasks, an area where user practices are not yet well established. To capture the nuances of workflow choices, reasoning processes, and human–AI interaction, we

adopt a qualitative research approach. This choice aligns with recent reflections in empirical software engineering, where researchers argue that AI tools introduce new socio-technical dynamics that cannot be fully understood through quantitative output measures alone (Treude and Storey, 2025; Trinkenreich et al., 2025).

Qualitative inquiry allows us to analyze participant behavior in context, combining observable actions with participants' verbalized reasoning. By drawing on multiple data sources, including screen recordings, think-aloud protocols, observation notes, and post-task interviews, we can reconstruct how students frame testing problems, how they delegate tasks to the AI, and how they evaluate or modify AI-generated outputs. This multi-source approach is essential for investigating AI as an active agent in developer workflows while preserving attention to human agency.

Our methodology also responds to the challenges discussed in recent work on empirical studies of AI-assisted development, which emphasize the need for interpretive depth when examining AI-mediated processes (Treude and Storey, 2025; Trinkenreich et al., 2025). To support this, our analysis combines open and axial coding with thematic analysis, enabling structured interpretation across heterogeneous data sources while maintaining rigor in how participant experiences and actions are represented.

2.3 RQ1: What strategies do students adopt when incorporating generative AI into unit testing workflows?

For RQ1, we analyzed participant video recordings of the assignments using an open coding approach to identify the actions they performed. The think-aloud protocol further enriched our analysis by revealing participants' intentions and decision-making processes, which helped us interpret and categorize their actions meaningfully.

The first author, who also served as the observer during data collection, conducted the initial open coding (Williams and Moser, 2019) by detecting and labeling the specific activities performed by participants. In the subsequent axial coding (Williams and Moser, 2019) stage, we assigned these open codes to broader strategy categories. To assess the reliability of this classification, another author independently assigned the open codes to strategies. We then calculated Cohen's Kappa (Gisev et al., 2013) between the two raters, which yielded a value of 0.75, indicating substantial agreement. Following this, both raters held a consensus meeting to resolve disagreements and finalize the mapping of actions to strategies.

This coding process resulted in the identification of four core strategies (C1-C4); these strategies describe whether the participant or ChatGPT was the main responsible for ideation and/or implementation. In addition, we identified supporting categories such as documentation use and mental model building. While the detailed definitions of these strategies are presented in Section 3.2, we used the distribution of coded actions, along with the provenance of generated assertions, to determine each participant's *dominant strategy* for each assignment. This provided a structured foundation for interpreting the behavior of the participants in later analyses.

```
String result = YourClassName.initials("HelloWorld");
assertEquals("HW", result);
```

(a) Initial generated test

```
assertEquals("HW", YourClassName.initials("HelloWorld"));
```

(b) Test after asking ChatGPT to merge previously generated test cases into a single one, the inlining action was not explicitly asked by the participant

```
assertEquals("HW", InitialsUtils.initials("HelloWorld"));
```

(c) Test after renaming `YourClassName` to `InitialsUtils` using ChatGPT

```
assertEquals("H", InitialsUtils.initials("HelloWorld"));
```

(d) Final test after a manual correction of the expected value

Listing 1: Example of iterative test generation and refinement, done by one of our participants

```
@Test
void testBasics() {
    // Test with a simple sentence and default delimiter (space)
    String result1 = initials("Hello World");
    assertEquals("HW", result1);

    // Test with custom delimiter (comma)
    String result2 = initials("Alice,Bob,Charlie", ',');
    assertEquals("ABC", result2);
}
```

Listing 2: Example of test produced by one of the participants

2.3.1 Provenance Analysis

To support our analysis of participant strategies and assess ChatGPT's contribution to participants' final test suites, we implemented an automated provenance analysis pipeline. Although this study focuses on participant strategies rather than individual assertions, we use assertions as a practical unit for this provenance analysis, allowing us to determine whether a test originated from ChatGPT output or was written by the participant.

By automating the comparison of conversations and tests produced by participants, we aim to reproducibly infer when tests were generated by ChatGPT. Provenance analysis approaches have already been used to compare code artifacts and ChatGPT conversations, notably through text-based approaches (Przymus et al., 2024). However,

```java
@Test
void testInitials_String_charArray() {
    char[] array = null;
    assertNull(WordUtils.initials(null, array));
    assertEquals("", WordUtils.initials("", array));
    assertEquals("", WordUtils.initials("␣␣", array));
    assertEquals("I", WordUtils.initials("I", array));
    ...
```

Listing 3: Example of assertions sharing the same input through a variable `array`, from the apache/commons-lang test suite

due to the semantic nature of tests and their expected low structural diversity, we use a syntactic representation.

To illustrate some of the challenges and specificities of our provenance analysis, Listing 1 presents tests at different steps of an iterative refinement process, as done by one of our participants. The examples present the same test assertion, i.e., an assertion statement and all the other statements necessary to evaluate the assertion.

Listing 1a is semantically equivalent to Listing 1b, but structurally different. Listings 1b to 1d have the same syntactic structure, but different references and literals. Notably, after fixing the class name in Listing 1c, it compiles and the test passes. Thus, we choose to consider listings 1a to 1c to be equivalent. We also consider Listing 1d to be similar to the other three.

Additionally, these tests could also be regrouped in the same test method, such as done by another participant (Listing 2). Complementarily, we also considered test assertions sharing statements (e.g., a statement that initializes a variable that is used in multiple test assertions) such as shown in Listing 3. While our provenance analysis can cater for the aforementioned situation, none of our participants wrote tests with similar sharing of a variable. These examples illustrate what our participants did (or could have done) to produce unit tests, demonstrating the structural variability of produced tests, as well as, their semantic similarity. Consequently, it also demonstrates assertions we would consider equivalent, such that GPT generated assertions could be accurately matched against the test suite of a participant at the end of their assignment–independently of structural differences.

For our provenance analysis, we collected two distinct artifacts at the end of each session: the test files produced by the participants and the conversation with ChatGPT. The ChatGPT conversations are a sequence of alternating prompts (by the participant) and answers (generated by ChatGPT). To accurately analyze a substantial portion of our data, we needed to examine tests at the level of individual statements. This required isolating test assertions and collecting their related statements. From these, we identified and extracted the semantically relevant elements of each assertion, including the test inputs, expected values, and the result of calling the method under test. Thus, we reached beyond purely textual approaches and leveraged syntax trees to analyze the test code (Dilavrec and Zaidman, 2025; Le Dilavrec et al., 2023).

In order to infer the provenance of test assertions using a syntactic approach, we first captured the variability in tests created/generated by the participants and ChatGPT in syntactic patterns.

To define test assertion patterns that represent all assertions in the test files and conversations, we first synthesized detailed patterns and then generalized them through an iterative process. The initial synthesis involves parsing, then selecting all the code blocks (surrounded by curly braces to help isolate from ill-formed source code mixed with natural language), and synthesizing the corresponding patterns (ignoring the type identifiers and the values of literals to limit the initial number of patterns). Then, to generalize the patterns, we iteratively apply the following four steps: (i) we isolate the assertions (i.e., creating a new pattern for each selected assertion while removing all the other assertions in a single step). (ii) associate local variable declarations to their references syntactically through a purely syntactic approach and by comparing their names (which works on our data because in Java, a given label can only be declared once per method), (iii) remove remaining local variable declarations which could not be associated to the assertion, and (iv) selecting the shortest list of patterns with the least number of unresolved references representing all the initial patterns.

At the end of this process, we obtain a set of abstract test patterns representing and categorizing all the assertions we collected. For example, Listing 1a would be described as a local variable declaration followed by an assertion statement. While the other test assertions of Listing 1 would be captured by the same generalized pattern described as a single assertion statement. With these patterns, we are now able to infer and describe the provenance of test assertions, controlling the patterns considered, and capturing the test elements to obtain the results of our similarity analysis.

Once the exhaustive set of test patterns has been obtained, we proceed with the automated inference of the origin of the tests. This process takes place in three steps, also preceded by the separation of questions and answers from conversations: (i) parsing, (ii) selection of tests based on the previously extracted patterns, and (iii) search in the conversation in chronological order for the first match.

The structured test patterns and provenance analysis enabled us to determine the origin of test assertions across all participants. Using this information, we can determine their dominant strategy over the implementation dimension.

2.4 RQ2: How do students prompt generative AI?

For RQ2, we extracted and analyzed the conversations between participants and ChatGPT. Similar to our approach in RQ1, we coded the prompts participants created, categorizing them based on key aspects of their interaction with ChatGPT. Specifically, we broke down each prompt into three dimensions:

- **Type:** The main purpose of a prompt; this can also be considered as its corresponding use case. Examples include asking ChatGPT to explain code, generate test cases, or clarify syntax.

- **Context:** The information participants provided to ChatGPT to support their request. This could include elements such as the source code file of the task, code documentation, an example test case, or a conceptual scenario for testing.
- **Demand:** The specific output the participant was requesting from ChatGPT, such as a test case, an explanation for a piece of code, or a refactored version of a provided code snippet.

This breakdown helps us understand what participants ask ChatGPT to do and how they construct their interactions. By distinguishing between **Type**, **Context**, and **Demand**, we can identify patterns in how students leverage ChatGPT, uncover their preferred use cases, and observe whether their prompting strategies evolve over time. Additionally, this categorization enables us to assess whether students are effectively providing the necessary context to receive high-quality responses and whether their demands align with their actual needs.

2.5 RQ3: What are the benefits and challenges of a generative AI-assisted test workflow for students?

To answer RQ3, we performed a thematic analysis (Willig and Rogers, 2017) of participants' post-task interviews and triangulated these insights with our observational data. Our goal was to understand both perceived and observed experiences with ChatGPT in unit testing workflows. We analyzed the interviews and observations to identify recurring patterns related to benefits and challenges, producing distinct codes that capture how students interacted with ChatGPT throughout the tasks.

3 Results

This section presents the findings of our study, structured to align with the research questions. Section 3.1 provides an overview of general quantitative metrics of the test suites that the participants engineered, including test coverage, assertion count, and task duration. Section 3.2 addresses **RQ1** by identifying the strategies students employed when integrating ChatGPT into their unit testing workflows. Section 3.3 answers **RQ2** through a detailed analysis of participants' prompts, examining their types and structure. Finally, Section 3.4 focuses on **RQ3** by outlining the benefits and drawbacks of AI-assisted testing. Across all sections, we draw from multiple qualitative and quantitative data sources to provide a comprehensive view of participant behavior and reasoning.

3.1 General Quantitative Metrics

Before turning to the qualitative analysis that is needed to answer our research questions, we begin with an overview of general quantitative metrics to contextualize participant performance. Specifically, we report these metrics across three groups: the student participants in our study, the reference implementation in the Apache.utils code base, and the participants in the prior study by Aniche et al. (2022). A

detailed breakdown of these metrics, including per-participant coverage scores, test and assertion counts, and task durations, is provided in Table 1.

To evaluate test effectiveness, we use mutation score, a widely adopted metric that captures the proportion of artificially injected faults—*mutants*—that a test suite detects (Jia and Harman, 2010). Mutation score is computed as $\frac{\text{Killed Mutants}}{\text{Total Mutants}} \times 100\%$. In our study, mutants were generated and executed using the PIT mutation testing tool with its default mutation operators, consistent with the configuration used by Aniche et al. (2022).

We observe that mutation score varies across the three groups. The participants of Aniche et al.'s study achieved the lowest average mutation score at 79.00%, with a relatively high standard deviation of 12.51%, indicating substantial variability (ranging from 62% to 92%). In contrast, the Apache.utils code base has the highest average mutation score at 91.00%, with a low standard deviation of 5.77%, showing consistent test effectiveness (ranging from 86% to 96%). Our participants fell in between, with an average mutation score of 88.96% and a moderate standard deviation of 6.89, ranging from 76% to 96%.

We also see that the number of tests engineered varies notably. Developers in Aniche et al.'s study created an average of 9.17 test cases per participant (ranging from 6 to 13), while the Apache.utils code base had far fewer test cases, averaging only 1.50 (although they contain many assertions). Our participants, by contrast, exhibited the highest number of tests created, averaging 12.08 test cases per participant, with a wide range from 1 to 21.

3.2 Strategies

Axial coding of participant activity data identified four distinct strategies, located along two dimensions. The first dimension is where the **idea** for a test case originates from, i.e., either from generative AI, or from the participant. The other dimension is **implementation**. Upon having the idea of what to test for, the code is written either by the generative AI or the participant. Although our focus is on ways in which participants interacted with generative AI, we also include a fourth strategy that captures cases where participants explicitly chose not to interact with AI. We consider this a strategy because it reflects a deliberate approach to completing the task in an AI-mediated environment. Therefore, the four different core strategies are:

- **C1** ($\mathbf{GPT_{IDEA}GPT_{IMPL}}$): This strategy refers to actions where the **idea** for the test case originates from **generative AI**, and the **implementation** is also done by Generative AI. Participants in this category rely on AI-generated test cases and AI-generated implementations. Example actions included "asking AI to generate test cases", "running and fixing generated test case".
- **C2** ($\mathbf{GPT_{IDEA}}\boldsymbol{P_{IMPL}}$): In this strategy, the **idea** for the test case is generated by **generative AI**, but the **implementation** is done by the **participant**. This indicates that while AI assists in generating test case ideas, participants take control of writing the actual test code. An example action is "implementing manual test case from GPT suggestion".

Table 1: Per-participant testing metrics across assignments.

Participant	Task	Mutation	Line Cov.	Branch Cov.	Tests	Assertions	Duration
P1	Initials	96%	100%	100%	7	18	44'42"
P2	Initials	96%	100%	100%	8	11	13'32"
P3	Initials	96%	100%	100%	1	11	17'03"
P4	Initials	96%	100%	100%	4	9	19'34"
P5	Initials	96%	100%	95%	12	12	29'16"
P6	Initials	96%	100%	100%	17	17	37'01"
P7	Initials	96%	100%	100%	15	16	35'52"
P8	Initials	96%	100%	100%	19	19	41'15"
P9	Initials	96%	100%	100%	14	14	18'46"
P10	Initials	87%	95%	95%	8	8	22'29"
P11	Initials	91%	100%	100%	9	9	24'02"
P12	Initials	96%	100%	100%	16	16	12'14"
P1	SubstringsBetween	86%	100%	100%	5	17	35'27"
P2	SubstringsBetween	86%	100%	95%	16	30	25'12"
P3	SubstringsBetween	86%	100%	95%	1	14	16'45"
P4	SubstringsBetween	81%	100%	90%	11	12	13'26"
P5	SubstringsBetween	76%	96%	95%	10	18	26'15"
P6	SubstringsBetween	76%	100%	100%	17	17	21'56"
P7	SubstringsBetween	86%	100%	100%	17	21	42'31"
P8	SubstringsBetween	86%	100%	100%	21	47	37'43"
P9	SubstringsBetween	86%	100%	100%	15	15	33'05"
P10	SubstringsBetween	81%	100%	100%	17	22	30'49"
P11	SubstringsBetween	86%	100%	100%	13	14	20'26"
P12	SubstringsBetween	81%	100%	95%	17	17	19'53"
Original1	Initials	96%	100%	100%	2	79	–
Original1	SubstringsBetween	86%	100%	100%	1	24	–
Original2	Initials	96%	100%	100%	2	75	–
Original2	SubstringsBetween	86%	100%	100%	1	24	–
Aniche1	Initials	92%	–	100%	11	–	56'34"
Aniche3	Initials	76%	–	91%	6	–	30'15"
Aniche7	Initials	84%	–	100%	12	–	40'53"
Aniche10	Initials	92%	–	91%	13	–	26'15"
Aniche2	SubstringsBetween	68%	–	93%	7	–	27'54"
Aniche13	SubstringsBetween	62%	–	75%	6	–	18'38"

Table 2: Category Mapping of Ideas and Implementations

Category Name	**Idea**	**Implementation**
C1 ($\mathbf{GPT_{IDEA}GPT_{IMPL}}$)	ChatGPT	ChatGPT
C2 ($\mathbf{GPT_{IDEA}}P_{IMPL}$)	ChatGPT	Participant
C3 ($\mathbf{P_{IDEA}GPT_{IMPL}}$)	Participant	ChatGPT
C4 ($\mathbf{P_{IDEA}}P_{IMPL}$)	Participant	Participant
Mental	-	-
Documentation	-	-
NA	-	-

- **C3 ($\mathbf{P_{IDEA}GPT_{IMPL}}$)**: This strategy involves the **participant** coming up with the test **idea**, but the **implementation** is carried out by **generative AI**. Here, participants rely on AI to generate test code based on their own conceptualization of the test case. An example action is "asking ChatGPT to refactor test suite".
- **C4 ($\mathbf{P_{IDEA}}\boldsymbol{P_{IMPL}}$)**: In this strategy, both the **idea** and the **implementation** of the test case are handled by the **participant**. This approach is entirely manual, with no reliance on AI for either test design or coding. Example actions included "implementing manual test cases", "running and fixing implemented test cases".

These four core strategies define the main ways in which participants interact with generative AI during the testing process, ranging from full AI dependence (C1-$\mathbf{GPT_{IDEA}GPT_{IMPL}}$) to fully manual testing (C4-$\mathbf{P_{IDEA}}\boldsymbol{P_{IMPL}}$).

3.2.1 Example Prompts

While the strategies above encompass a broad range of participant actions (e.g., reading documentation, validating results, refining outputs), we include a set of prompt examples to illustrate how these strategies manifested in prompts to ChatGPT. These examples show the type of input participants provided, but they do not exhaustively describe all actions within each strategy.

C1 — $\mathbf{GPT_{IDEA}GPT_{IMPL}}$

E1 (Participant 1). *"Do you think there are other cases that I should test for when testing the `initials` method?"* Context: full Java class and current test suite. Demand: additional test ideas and AI-generated implementations.

E2 (Participant 9). *"Examine the above code. 1. Does the implementation match its purpose given in the docstring? 2. Give me a JUnit test suite for the above code. Cover as many edge cases as possible. Separate assertions into test cases (one assertion per test!)."* Context provided: full Java class. Demand: full JUnit test suite in a specific format.

C2 — $\mathbf{GPT_{IDEA}}\boldsymbol{P_{IMPL}}$

E3 (Participant 5). *"I will give you a method; please give me (in pseudocode) examples of test cases."* Context provided: Java class code (no Javadoc). Follow-up: participant manually implemented the suggested tests.

C3 — $\mathbf{P_{IDEA}GPT_{IMPL}}$

E4 (Participant 2). *"Can you now write tests for the following cases: 1. The open and close tags are not in `str`. 2. There is only one match. 3. The open tag is at the start. 4. The close tag is at the end of `str`. 5. The string contains an open tag that is almost correct, but the last character does not match."* Context provided: test case ideas; JUnit setup and method handle were provided earlier.

In addition to the four core strategies, we identified 3 supporting categories that capture auxiliary actions performed by participants. These categories are Documentation, Mental, and Other:

- **Documentation**: This category encompasses instances where participants engage with the provided Javadoc comments in the assignment source code file to familiarize themselves with the assignment code. It involves reading and interpreting this documentation. An example action is "reading documentation".
- **Mental**: This refers to moments where participants are actively thinking about the code, analyzing its structure, or reading through the code to develop a mental model for understanding. This cognitive process helps participants break down and internalize the logic of the code. Example actions are "looking at method code body and precondition" and "considering completeness of test suite".
- **Other**: These activities include actions taken by participants that are not directly related to task advancement but are instead aimed at resolving personal difficulties, such as asking for clarification on a method or posing a question about Java syntax. Since these actions do not influence the participant's overall strategy and are specific to their individual understanding, we exclude them from the strategy discussion. An example action is "asking ChatGPT about Java syntax".

Figure 3 provides a detailed breakdown of participants' actions during the experiment, organized per assignment (e.g., P3i refers to participant 3's task involving the *initials* method, while P3s refers to the task involving the *substringsBetween* method). Now that we have explained all strategies (C1–C4) and auxiliary actions (documentation, mental, and other), we turn to identifying the dominant component of a participant's strategy for creating unit tests. An intuitive approach is to observe the most frequently occurring strategic action, either in terms of the number of actions or the total duration spent on that action throughout the task. However, a major caveat with this type of categorization is the disparity in terms of produced test cases between strategies that involve automated test creation (C1-$\mathbf{GPT_{IDEA}GPT_{IMPL}}$) and those that involve fully manual test creation (C4-$\mathbf{P_{IDEA}}\boldsymbol{P_{IMPL}}$).

For example, if a participant performs both C1 and C4 actions twice, the resulting test suite might still be primarily composed of generated tests. This suggests that automated actions can have a larger impact on the test suite compared to manual ones. To account for this, we first assess the direct contribution of ChatGPT to the participants' final test suites by figuring out which assertions present in the test suite were suggested by ChatGPT, allowing us to evaluate the actual influence of generated tests. This analysis provides a measure of the participant's reliance on ChatGPT.

We use this ChatGPT reliance to distinguish between strategies dominated by automated actions (C1 and C3) and those driven more by manual efforts (C2 and C4). Following this initial separation, we further differentiate these strategy groups by examining the frequency of performed actions. This process enables us to assign a dominant strategy to each participant, or in cases where no single strategy dominates, to classify them as employing a mixed strategy.

To infer the origin of a test as suggested by ChatGPT, we relied on the structure of the conversations and on the patterns we extracted (see Section 2.3.1). For each test file and its corresponding ChatGPT conversation, we extracted the test assertions along with their related statements. Each assertion in the test file was then iteratively compared to each assertion in the conversation in chronological order. If we find a match

Fig. 3: Participant Approaches to Testing Tasks

in an answer generated by ChatGPT, then we consider the assertion as generated. If we do not find a match, we consider the assertion to be manually implemented.

Table 3 presents the results of our analysis and enables us to compute the overall percentage of assertions that we inferred to originate from the ChatGPT conversations. This table also provides the number of test assertions present in the test files.

Additionally, based on our analysis of test patterns, we also provide the breakdown of a subset of patterns representing the examples in Listing 1 where the assertion requires from zero to any number of local variable declaration and we only compare the inputs (ignoring the expected value).

We omitted the breakdown of more complex and less frequently occurring test patterns, such as ones requiring for loops or assignment statements. In particular, we did not analyze ∼10% of the test assertions in the test files of our participants. Participant 1 (P1) represents

7.6% (31 of 404) of non-detailed patterns, which is due to P1 mostly using parametric testing, thus separating the assertions from their inputs across different methods and making them difficult to analyze. The remaining non-detailed assertions only represent 0.3% (13 of 404) of the total.

We present the detailed provenance ratios as fractions of the number of assertions originating from ChatGPT over the total number of assertions in the test file for the given pattern. The detailed provenance ratios are presented from instances with no local variable declaration (a single assertion statement) to five, where a match only occurs between instances with the same number of local variables. On the first column of the breakdown, the single statement test assertions represent 46% of assertions in test files. The last column of the breakdown presents the result of combining all the previous patterns, such that additional matches in this column represent instances where a test assertion was refactored to use a different number of local variables. The details and intermediate results can be found in our data package (Ardic et al., 2025b).

C1 ($\mathbf{GPT_{IDEA}GPT_{IMPL}}$) dominant:

Participants **P3, P8, P11, and P12** consistently relied on ChatGPT for both assignments. **P4**, on the other hand, initially employed a **C1** dominant strategy but transitioned to a mixed strategy for the second assignment. **P9** did the opposite and moved from a mix to a **C1** dominant approach. As a result, we observed that about half of the participants primarily utilized **C1** in at least one of their assignments.

C4 ($\mathbf{P_{IDEA}}P_{IMPL}$) dominant:

This was the second most common strategy. Traditional programming was the preferred approach for **P1, P5, and P7** in both assignments, as well as in the first assignment for **P10**. **P12** was the only participant who did not engage in any **C4**-type actions.

C2 ($\mathbf{GPT_{IDEA}}P_{IMPL}$) dominant:

This is the third most common strategy, following **C1** and **C4**. It was the dominant approach in the second assignment for **P10**. Additionally, it appeared in the assignments of **P1**, **P5**, and **P7**, though it was secondary to **C4**.

C3 ($\mathbf{P_{IDEA}GPT_{IMPL}}$) dominant:

This strategy was the least frequently used, emerging as dominant only in the second assignment for **P2**. It was also observed in the second assignment of **P8** and **P4**, as well as the first assignment of **P9** and **P2**.

Table 3: Participant Strategy and Assertion Percentages

Assignment	Strategy	Assertions per #local var. 0	1	2	3	4	5	≤5	Assertion count	% gen
P12s	C1 ($\mathbf{GPT_{IDEA}GPT_{IMPL}}$)	17/17						17/17	17	100%
P12i	C1 ($\mathbf{GPT_{IDEA}GPT_{IMPL}}$)	16/16						16/16	16	100%
P11s	C1 ($\mathbf{GPT_{IDEA}GPT_{IMPL}}$)	5/10	4/4					9/14	14	64.3%
P11i	C1 ($\mathbf{GPT_{IDEA}GPT_{IMPL}}$)	1/1	1/2	2/4	2/2			6/9	9	66.7%
P10s	C2 ($\mathbf{GPT_{IDEA}}P_{IMPL}$)				0/11	0/11		0/22	22	0.%
P10i	C4 ($\mathbf{P_{IDEA}}P_{IMPL}$)			1/8				1/8	8	12.5%
P9s	C1 ($\mathbf{GPT_{IDEA}GPT_{IMPL}}$)	13/15						13/15	15	86.7%
P9i	Mixed	12/14						12/14	14	85.7%
P8s	C1 ($\mathbf{GPT_{IDEA}GPT_{IMPL}}$)		24/47					24/47	47	51.1%
P8i	C1 ($\mathbf{GPT_{IDEA}GPT_{IMPL}}$)	4/10		6/7	2/2			12/19	19	63.2%
P7s	C4 ($\mathbf{P_{IDEA}}P_{IMPL}$)	0/6	0/10			0/1	0/4	1/21	21	4.8%
P7i	C4 ($\mathbf{P_{IDEA}}P_{IMPL}$)	0/15	0/1					0/16	16	0.%
P6s	Mixed	9/17						9/17	17	52.9%
P6i	Mixed	9/14	1/1					10/15	17	33.3%
P5s	C4 ($\mathbf{P_{IDEA}}P_{IMPL}$)	0/7	0/1	0/6				0/14	18	0.%
P5i	C4 ($\mathbf{P_{IDEA}}P_{IMPL}$)	0/11						0/11	12	0.%
P4s	Mixed		7/12					7/12	12	58.3%
P4i	C1 ($\mathbf{GPT_{IDEA}GPT_{IMPL}}$)	0/1	7/8					7/9	9	77.8%
P3s	C1 ($\mathbf{GPT_{IDEA}GPT_{IMPL}}$)	9/10						9/10	14	64.3%
P3i	C1 ($\mathbf{GPT_{IDEA}GPT_{IMPL}}$)	8/10						8/10	11	72.7%
P2s	C3 ($\mathbf{P_{IDEA}GPT_{IMPL}}$)	5/7	13/20		0/1	1/1		19/29	30	63.3%
P2i	Mixed		1/3	5/8				6/11	11	54.5%
P1s	C4 ($\mathbf{P_{IDEA}}P_{IMPL}$)	0/1						0/1	17	0.%
P1i	C4 ($\mathbf{P_{IDEA}}P_{IMPL}$)	0/2	0/1					0/3	18	0.%

Mixed:

Participants **P2, P4, P6, and P9** exhibited mixed strategies in their first assignments, where no single approach was clearly dominant. The most frequently observed combination was the **C1-C4** pair.

Documentation:

The documentation category is primarily observed at the beginning of assignments for most participants. However, it is absent for P3 and does not appear in the first assignment for P5, though it is present in the second. In general, documentation-related actions occur early in the assignment and are almost always within the first half. Most participants interact with the Javadoc only once, with a maximum of three interactions. This suggests that participants prioritize reviewing documentation at the start of their tasks, aligning with findings from Aniche et al. (2019).

Mental:

The mental category is present for all participants. The proportion of time spent on this activity, relative to the overall assignment duration, varies between participants, ranging from 8% to 40%. Regardless of the dominant strategy for creating their test suites, we observed that participants attempted to build a mental model of code.

Summary for RQ1:

We observed four core strategies (C1-$\mathbf{GPT_{IDEA}GPT_{IMPL}}$ to C4-$\mathbf{P_{IDEA}}\boldsymbol{P_{IMPL}}$) characterizing how participants engaged with ChatGPT during unit test creation, defined along two dimensions: the origin of the test case idea and the implementation source. These strategies range from full reliance on ChatGPT (C1-$\mathbf{GPT_{IDEA}GPT_{IMPL}}$) to fully manual workflows (C4-$\mathbf{P_{IDEA}}\boldsymbol{P_{IMPL}}$). To determine each participant's dominant approach, we combined strategy frequency with automated provenance tracing of test assertions. This allowed us to assess not just action frequency but also the concrete contribution of ChatGPT to the final test suites. In addition to these strategies, we identified three auxiliary activity types: documentation, mental modeling, and non-task-related actions, that contextualize participant behavior beyond direct test creation. Together, these findings offer a comprehensive account of participant strategies and levels of ChatGPT reliance.

3.3 Prompts

We investigated a total of 111 prompts submitted across all assignments. On average, participants wrote 9.25 prompts per assignment. The number of prompts per participant ranged from a minimum of 3 to a maximum of 17, reflecting variation in how students used ChatGPT across tasks. This prompt volume provides a rich basis for investigating how students engaged with ChatGPT on what types of requests they made, how much context they provided, and the overall structure of their interactions.

To analyze the participant prompts during the assignments, we have divided each prompt into three components: type, context, and demand. These are visualized in Figure 4. These components help us understand how students interact with ChatGPT during unit testing tasks, including why they interact with the tool (type), what they ask for (demand), and what information they provide to it (context).

3.3.1 Prompt Types

The prompt types reflect the general use cases. The dominant type was **test generation** (72 out of 111), indicating that the primary use case for ChatGPT was to assist with the creation of unit tests. Other uses included **explaining** code or test behavior (17), dealing with **syntax-level** concerns (15), and performing **refactorings** on existing test code (7). This breakdown reveals that ChatGPT's primary role was in test generation, but that it also served as a flexible assistant throughout the assignments.

3.3.2 Prompt Demand and Test Generation Strategies

Prompt demand captures the participants' goals when interacting with ChatGPT. The three dominant categories were:

- **Explain** (33): These prompts sought clarification about various aspects of the code or testing process, including syntax, behavior, and results.

  Within this category, the most common explanations requested were related to **code or method behavior** (14 instances) and **syntax** (13 instances). Less frequent but

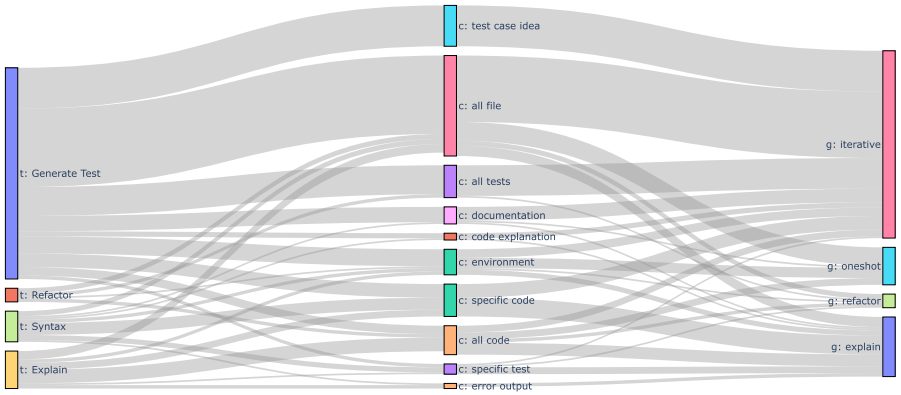

Fig. 4: Breakdown of participants' prompts by type (t), provided context (c), and grouped demand (g), where g reflects consolidated output requests

Table 4: Prompt Demand Group Breakdown by Participant-Assignment Pair

Participant	Explain	One-shot	Iterative	Refactor
P1i	8	0	1	0
P2i	0	0	2	0
P3i	3	1	3	2
P4i	2	1	1	1
P5i	0	1	1	0
P6i	2	1	0	0
P7i	3	0	1	0
P8i	1	0	5	0
P9i	0	1	0	1
P10i	1	1	2	0
P11i	3	0	8	0
P12i	0	1	1	0
P1s	1	0	2	0
P2s	0	0	7	0
P3s	2	1	3	2
P4s	0	1	0	1
P5s	0	1	0	0
P6s	3	2	3	0
P7s	1	0	6	0
P8s	0	1	4	0
P9s	0	1	2	0
P10s	1	0	2	0
P11s	2	1	3	0
P12s	0	1	1	0

notable were prompts asking for explanations of **code output** (2) or **test results** (2), indicating a need to interpret what the system produced in response to prior inputs.

- **Iterative test generation** (71): These involved generating individual test cases or small groups of tests in a step-by-step fashion.

  Within the category of iterative test generation, we identified several specific types of demands that reflect how students used ChatGPT to build and refine their test suites progressively. The most frequent demands were to **generate specific test case** (28 instances) and **expand an existing test suite** (16 instances), highlighting a common strategy of incrementally improving coverage based on previously existing tests, which were either generated, manually written or both.

  Participants also requested the generation of **edge cases** (9). Four prompts asked for **ideas for test cases**, indicating that participants were also interested in conceptual descriptions of tests they intended to generate or implement.

  Other iterative demands included requests for a **specific test format** (5) and a **specific number of tests to generate** (6). Though less frequent, additional demands included generation of **example test inputs** (2) and explicit **non-edge case** scenarios (1).

  These patterns indicate that students using iterative prompting often approached ChatGPT as a collaborator, using it to create the test suite in a structured, layered manner rather than through one-shot generation. This led to participants submitting more prompts as demonstrated in Table 4.

- **One-shot test generation** (16): These prompts asked ChatGPT to generate a full test suite in a single prompt.

  One-shot prompts are fewer in number, as demonstrated in Table 4. This is expected because each one-shot prompt delivers an entire test suite, whereas iterative prompting divides the process into multiple, smaller requests. Thus, the lower frequency of one-shot prompts should not be interpreted as a lack of interest, but rather as a reflection of a different engagement style.

3.3.3 Prompt Context

The prompt context refers to the kind of information participants included with their prompts. Out of a total of 110 prompts, participants included a wide variety of contextual elements to help the model generate more relevant and accurate outputs.

The most common form of context was **all file** inclusion (57 instances), where participants supplied the entire source file with the JavaDoc documentation, to give ChatGPT a comprehensive understanding of the codebase. This was followed by **test case ideas** (24), indicating that some participants preferred to describe what the test to be generated should do, rather than relying on providing source code.

Participants also frequently provided **specific code** snippets (19) or included **environment** details (15) like the programming language or unit test framework they used (Java and JUnit). In 19 prompts, participants also included **all tests** from their current test suites at the time of prompting. This was typically done toward the end of the interaction, either to help ChatGPT iteratively refine the test suite or to assess its completeness. In 9 prompts, participants provided a **specific test** or a subset of tests

from their suite. These were most often provided as examples to guide the generation of similar test cases or to demonstrate a preferred format or template.

Less frequently, students included **documentation** (10), **code explanations** (4), **error outputs** (3), and **specific test cases** (6). These more focused forms of context show that some students were attempting to guide ChatGPT more precisely to resolve specific issues such as debugging, troubleshooting, or obtaining detailed clarifications.

Overall, this diversity in context types reflects flexible prompting strategies. While some students opted for completeness by providing full files, for example, providing the entire task file, which includes code under test, auxiliary methods, and Javadoc, others took a more targeted approach, sharing code snippets or specific test case ideas, guiding ChatGPT with precise information. Test case ideas, in particular, indicate that students often engaged with ChatGPT at a conceptual level, describing what should be tested rather than only providing code under test.

3.3.4 One-shot vs. Iterative Prompting Strategies

To determine whether distinct patterns exist in how participants interact with ChatGPT, we constructed state diagrams for each participant-assignment pair. These diagrams are available in our data package (Ardic et al., 2025b). These diagrams visualize the sequence and structure of prompts, enabling us to identify systematic differences in engagement styles. Through this analysis, we inductively derive two main prompting strategies: **one-shot**, where participants requested a complete test suite in a single prompt, and **iterative**, where test generation was approached through a sequence of smaller, focused prompts.

To avoid confusion with standard prompting terminology (e.g., zero-shot or few-shot prompting, which describe how many example outputs are included in a prompt (Al Nazi et al., 2025)), we clarify that in our study "one-shot" and "iterative prompting" refer instead to *how participants structured the task of generating a test suite*. Our use of "one-shot" denotes asking for a complete test suite in a single request, whereas "iterative prompting" denotes breaking the task into multiple smaller prompts and refinements.

We further categorized participants based on their test generation strategy: **one-shot** vs. **iterative**. A breakdown of this categorization is available in Table 5.

This division reveals that half of the participants (6 out of 12) followed a one-shot test generation approach, treating ChatGPT as a tool for producing complete solutions. In contrast, 3 participants used iterative prompting, engaging in a more interactive and gradual process where ChatGPT's outputs were actively refined and validated over time. Additionally, the remaining 3 participants (P8, P10, and P11) exhibited mixed behavior, using different strategies across assignments.

Summary for RQ2:

Students engaged with ChatGPT in multifaceted ways, using it not only for generating test cases but also for explaining logic, clarifying syntax, and refining outputs. Most prompts were rooted in meaningful contexts, and participants exhibited a variety of prompting styles; some preferred quick, one-shot generation, while others engaged

Table 5: Participant-Assignment pairs by prompting strategy.

Iterative Strategy	One-shot Strategy
P1i, P1s	P3i, P3s
P2i, P2s	P4i, P4s
P7i, P7s	P5i, P5s
P8i	P6i, P6s
P10s	P8s
P11i	P9i, P9s
	P10i
	P11s
	P12i, P12s

in more iterative and collaborative workflows. While the short nature of the assignments limits our ability to track prompt evolution over time, the data suggests that students were mostly consistent in how they used ChatGPT during these assignments.

Table 6: Summary of Perceived Drawbacks and Benefits

Category	Count
Benefits	
Cognitive Load	10
ChatGPT as Supervisor	6
Participant as Supervisor	8
Saves Time	13
Test Generation	6
ChatGPT Expands Search on New Subject	1
Uses ChatGPT for Knowledge Questions	1
ChatGPT Explains Methods Well	1
Uses ChatGPT for Syntax	3
Drawbacks	
Lack of Ownership	6
Lack of Quality	13
Lack of Trust	9
Loss of Skill	1
Negative Belief	1
Prevent Learning	1

3.4 Post Task Interviews

To address **RQ3** — *What are the benefits and challenges of a generative AI-assisted test workflow for students?*, we conducted a thematic analysis of participants'

post-task interviews, supported by observational data. This analysis revealed recurring patterns in how students interacted with ChatGPT during the testing tasks.

Several benefits and drawbacks emerged from this investigation, particularly regarding productivity, support, ownership, and the nature of collaboration between participants and ChatGPT. These are summarized in Table 6, which reports how many times each theme was mentioned during the interviews. A single participant may mention the same benefit or drawback multiple times across different interview questions.

3.4.1 Benefits

The following themes capture the key ways in which students found ChatGPT beneficial during their unit testing tasks. These benefits range from gains in efficiency and reduced mental effort to more subjective forms of support and collaborative interaction.

Saves Time

Time-saving was the most frequently mentioned benefit (13 mentions). Participants found that ChatGPT accelerated their workflow by quickly generating boilerplate or initial versions of test cases, allowing them to iterate faster or redirect time to more critical thinking tasks. This aligns with the general perception of AI as a productivity enhancer.

Reduced Cognitive Load

Participants frequently reported that ChatGPT helped reduce their **cognitive load** (10 mentions). By offloading some repetitive aspects of test creation, students could focus more on understanding the application logic or coming up with more test cases after covering the basic functionality of the code under test quickly. This can make unit testing less mentally taxing if generative AI use does not detriment personal skill development, especially for less experienced programmers.

This aligns with findings by Camilleri et al. (2022), who observed that test case design is one of the testing tasks associated with significant cognitive workload, especially among less experienced software testers. Using the NASA-TLX instrument, this study showed elevated levels of mental demand, effort, and frustration in such tasks, confirming that reducing the load associated with test case creation could offer practical benefits.

Supervision Dynamics – Human and AI

We identified two distinct yet complementary supervisory dynamics that capture the participants' overall approach to collaborating with AI for unit testing. Some participants perceived **ChatGPT as a supervisor** (6 mentions), guiding them through the unit testing process, suggesting test logic, or explaining unfamiliar concepts. Conversely, others viewed themselves as the supervisor of the AI (**Participant as Supervisor**, 8 mentions), feeling empowered to critique, refine, or selectively use ChatGPT's suggestions. This reflects a more interactive and agent-driven relationship with the tool.

Test Generation Assistance

ChatGPT's ability to assist with **test generation** (6 mentions) was highlighted as a key benefit. While concerns about test quality persist, many students appreciated having a starting point from which they could build or improve test cases. The tool serves more as a productivity aid than a complete solution, as the generated tests frequently contain errors in setting up the expected values for assertions.

Targeted Use Cases

Some participants reported less frequent use cases of ChatGPT: It was used much like a search engine to **expand their search** for new information (1 mention), to answer **knowledge-based questions** (1 mention) such as understanding testing concepts. Participants also used it to explain methods or APIs clearly (**ChatGPT explains methods well**, 1 mention) and to address **syntax-level** challenges (3 mentions), helping overcome low-level barriers that might otherwise stall progress.

These niche benefits demonstrate how ChatGPT can flexibly integrate into different parts of a student's workflow, addressing both high-level reasoning and low-level mechanics.

3.5 Drawbacks

While AI-assisted tools like ChatGPT offer several perceived benefits in the testing workflow, participants also reported a range of concerns and limitations. These drawbacks provide critical insights into the potential risks of over-reliance on automation and underscore the need for careful integration of AI into learning environments. Similar concerns were also emphasized by Choudhuri et al. (2024b).

Lack of Ownership

Several participants (6 mentions) expressed a **lack of ownership** over the test cases generated by ChatGPT. For the tests that were not written manually, students felt less accountable for their correctness and were more inclined to accept the output without critical evaluation. This detachment can reduce opportunities for active learning and reflection, both of which are essential for developing a strong understanding of unit testing principles. Prior work has similarly noted that perceived ownership tends to increase when learners are more actively involved in generating solutions or when they are reminded of their involvement through prompting and refinement (Wasi et al., 2024).

Lack of Quality

The most frequently cited drawback (13 mentions) was the **lack of quality** in AI-generated test cases. Participants noted that these tests often:

- Miss edge cases or complex logic.
- Do not align well with functional requirements.
- Are unpolished and contain duplicates.

This suggests that while ChatGPT can generate syntactically correct test cases, human oversight is essential to ensure relevance and depth. The quality issues also raise concerns about students being misled into thinking their testing is more thorough than it actually might be.

Lack of Trust

A related but distinct concern was a **lack of trust** in ChatGPT's output (9 mentions). Even when the tool produced runnable test cases, participants often second-guessed the correctness or coverage of the results. Key issues included:

- Inconsistencies in generated outputs across similar prompts.
- Test cases that passed trivially without real validation.
- Redundant or irrelevant tests that added noise rather than insight.

This skepticism can be constructive if it encourages critical review, but it may also increase cognitive load and reduce confidence in using AI tools effectively.

Loss of Skill and Prevented Learning

Some participants feared that using ChatGPT might lead to a **loss of skill** or **prevent learning** (1 mention each). These responses, while less frequent, signal a concern that automation might short-circuit the learning process. When students bypass the logic-building and problem-solving steps by outsourcing them to AI, they may miss foundational concepts critical for future development tasks. This aligns with recent findings of Fan et al. (2025) that generative AI tools such as ChatGPT, while shown to be effective at improving short-term task performance, may not fully support deeper learning outcomes such as intrinsic motivation, knowledge transfer, or self-regulated learning, thus fostering dependence on technology.

4 Discussion

In this section, we interpret and contextualize our findings. We begin by examining how participants' prompting styles and strategic choices interacted, shedding light on the relationship between user control and AI assistance. We then discuss the qualitative and quantitative qualities of the resulting test suites, including test smells and assertion-level behavior. Finally, we reflect on broader themes such as control, cognitive load, and experience, and introduce a conceptual framework that captures how these factors might shape engagement in AI-assisted testing workflows.

4.1 Interactions Between Strategy and Prompting Approach

We categorized participants' strategies for unit testing within an AI-assisted workflow into four types, with C1-$\mathbf{GPT_{IDEA}GPT_{IMPL}}$ and C4-$\mathbf{P_{IDEA}}P_{IMPL}$ emerging as the most dominant approaches. We also examined how participants approached prompt creation for ChatGPT in the context of test generation.

Our key observation is that the primary difference across participants lies in how they obtained their test assertions: either by incrementally expanding the test suite

with each prompt using an iterative approach, or all at once via one-shot generation of a complete test suite followed by refinements.

To explore whether there was a relationship between participants' prompting approach and their overall testing strategy, we analyzed how frequently different strategies were used alongside iterative or one-shot generation. Here, strategy refers to how participants leveraged ChatGPT, ranging from heavy reliance (C1-$\mathbf{GPT_{IDEA}GPT_{IMPL}}$) to minimal usage with more manual implementation (C4-$\mathbf{P_{IDEA}}\boldsymbol{P_{IMPL}}$).

We hypothesized that participants who relied more heavily on ChatGPT (C1) would favor the one-shot generation approach, as it requires less manual involvement. This trend was supported in our data: as shown in Table 7, 9 out of 10 C1-dominant assignments used one-shot generation.

Moreover, participants following Strategy C4, who favored manual implementation with less AI involvement, more often used iterative prompting, which better aligns with their workflow. Two out of the three cases where C4 dominant participants used one-shot generation came from a single participant who applied it at the end of their assignment to compare results against their manually developed test suite.

Generation	C1($GPT_{IDEA}GPT_{IMP}$)	C2($GPT_{IDEA}P_{IMP}$)	C3($P_{IDEA}GPT_{IMP}$)	C4($P_{IDEA}P_{IMP}$)	Mixed
Iterative	1	1	1	4	1
One-shot	9	0	0	3	4

Table 7: Combinations of generation approach and strategies.

In interviews, 6 out of the 9 participants who used one-shot generation in at least one task said they would have been able to complete the tasks without relying on any external resources, including ChatGPT or Stack Overflow. However, in practice, many leaned toward strategies that reduced manual effort and handed off more responsibility to ChatGPT. The frequent use of one-shot generation, especially among those who relied heavily on the model, points to a preference for speed and convenience over fine-grained control. While this approach can be efficient, it also carries some risk, particularly if participants rely too heavily on AI-generated output without critically evaluating its content.

4.2 Code Smells and Comparative Code Quality

Naturally, in investigating participant strategies, we also wanted to see whether certain approaches led to more successful outcomes in terms of measurable qualities of the resulting test suites. While the previous section focused on how participants worked, particularly their prompting styles and reliance on ChatGPT, this section turns to what those workflows produced.

Specifically, we evaluated each test suite based on adequacy (Zhu et al., 1997) and effectiveness (Athanasiou et al., 2014; Petrović et al., 2021). Our goal was to determine whether prompting methods (iterative vs. one-shot prompting) or workflow strategies (C1–C4) had a noticeable impact on the quality or completeness of the resulting tests.

In Table 1, we present the test suite metrics for all participants. We examined line coverage and mutation scores as indicators of test adequacy and effectiveness. However, when comparing these results across different prompting styles or strategy groups, we found no clear patterns that could distinguish one approach from another. In other words, neither a prompting method nor a strategy appeared to consistently produce better-performing test suites based on these metrics.

That said, when compared to Aniche et al.'s prior study using the same tasks to analyze how professional developers engineer test cases (Aniche et al., 2022), our participants' test suites performed better in both line coverage and mutation scores. This suggests that having access to ChatGPT may offer a notable advantage in generating effective test cases. Nonetheless, within our own study, differences in prompting or workflow style did not translate into major differences in output quality.

To further investigate potential quality differences, we examined the presence of test smells (Greiler et al., 2013; Meszaros, 2007; Moonen et al., 2008; Spadini et al., 2018), which are patterns in test code that may indicate maintainability or reliability issues. We used TSDetect (Peruma et al., 2020) to automatically analyze the test suites. Out of the 19 test smells the tool can detect, 7 were observed in the participants' submissions. These smells, defined as follows in Peruma et al. (2020), provide insight into potential shortcomings in the test suites :

- **Assertion Roulette**, a test method that contains more than one assertion statement without an explanation/message (parameter in the assertion method)
- **Conditional Test Logic**, a test method that contains one or more control statements (i.e., if, switch, conditional expression etc.)
- **Eager Test**, a test method that contains multiple calls to multiple production methods
- **Lazy Test**, multiple test methods calling the same production method
- **Duplicate Assert**, a test method that contains more than one assertion statement with the same parameters
- **Ignored Test**, a test method or class that contains the `@Ignore` annotation
- **Magic Number Test**, an assertion method that contains a numeric literal as an argument

To explore whether any of the participant groupings were associated with systematically higher or lower occurrences of test smells, we conducted statistical comparisons across the identified strategies (C1 to C4) and prompting styles (iterative versus one-shot). We used the Mann–Whitney U test (McKnight and Najab, 2010) for these comparisons. Given the small sample size, our statistical tests did not reveal significant differences between strategy or prompting groups in the number of detected test smells, meaning that even if such differences exist, they could not be detected with the available data.

In summary, this finding suggests that for the unit testing tasks in our study, and provided that the participants differed in their approaches to the task, with varying levels of automation, control, and prompt structure, we did not observe clear differences in test quality in our data, as measured by the presence of test smells.

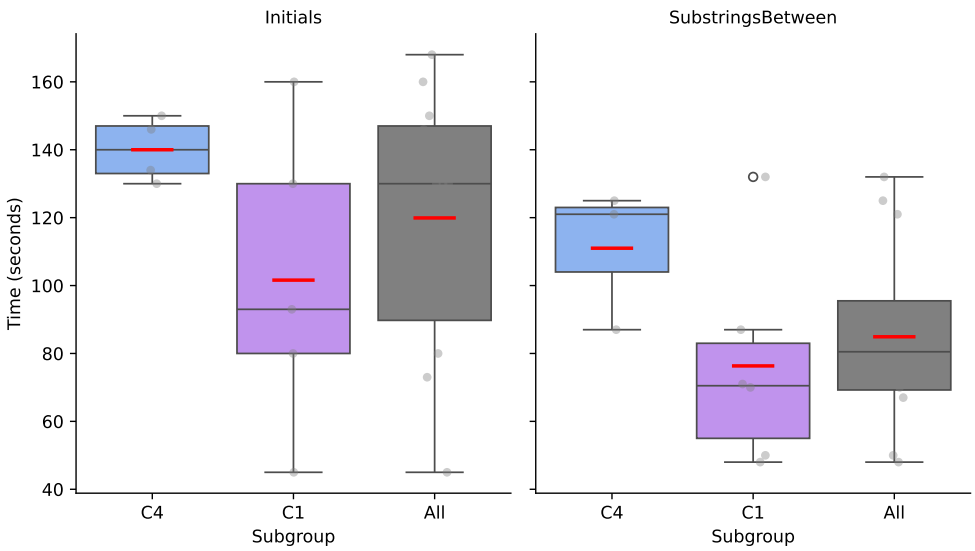

Fig. 5: Assertion completion time across assignments, grouped by participant strategy. The lines in red are average values (seconds)

4.3 Average Time to Write an Assertion

Figure 5 visualizes the time participants took to complete a single assertion for each assignment. While participants who relied more heavily on ChatGPT-assisted workflows, particularly those who followed the $\mathbf{GPT_{IDEA}GPT_{IMPL}}$ strategy, tended to complete assertions more quickly on average, the difference was not substantial enough to support the claim that ChatGPT use reduces the completion time for this task.

This suggests that the main benefit of ChatGPT-assisted workflows may lie more in reducing the cognitive or manual effort required, rather than providing a significant time advantage. The tasks, although not overly complex, are not trivial either and still require a level of reasoning and correction that ChatGPT cannot fully automate.

A notable pattern among ChatGPT reliant participants was the tendency to adopt a "one-shot" generation strategy, attempting to produce an entire test suite, including assertions, in a single prompt. While this approach could initially reduce the time spent writing, it often led to errors. We observed that all participants who used one-shot generation needed to manually modify the generated test cases. The majority of these were simple fixes, such as changing incorrect expected values in the generated assertions. Participants subsequently spent time identifying and correcting these issues, which decreased the net time savings.

```
/*
 * Test case to verify extraction of initial characters from a string
 * containing multiple words separated by a specific delimiter.
 */
```

```
@Test
public void testMultipleWordsWithCustomDelimiter() {
    assertEquals("CST", initials("Customer: Service: Team", ':'));
}
```

Listing 4: Example of GPT-generated test with incorrect assertion

Corrected version of the assertion:

```
assertEquals("C  ", initials("Customer: Service: Team", ':'));
```

In this example, ChatGPT incorrectly assumes that each capitalized word contributes its first character, overlooking that the `initials` method only takes the first character of the original string and the segments after the custom delimiter. As "Service" and "Team" follow colons but begin with whitespace, their initials result in two space characters. This subtle misunderstanding exemplifies the most common type of assertion error produced by ChatGPT. Assignment implementations are provided in our data package (Ardic et al., 2025b).

It is likely that the frequency and nature of such mistakes would vary across different tasks, making our findings on time savings difficult to generalize beyond this context. Nonetheless, these subtle assertion errors represent the primary barrier to fully automating test generation with ChatGPT in our setting. At the same time, our analysis suggests that the test ideas proposed by ChatGPT were generally sound, with participants frequently choosing to incorporate them into their final test suites. This indicates that while the implementation details often required correction or refinement, the conceptual value of ChatGPT's suggestions was high. As the capabilities of generative models continue to improve, it is plausible that these implementation errors will become less frequent, enabling greater automation in similar tasks. For now, however, the main advantage of using ChatGPT appears to lie in the reduced cognitive effort required to ideate and structure test cases, rather than in fully automating their execution.

4.4 Control and Cognitive Load in ChatGPT Use

Our study, focused on students as a relatively low-experience user group, investigates how individuals navigate ChatGPT-assisted workflows in software testing contexts. A key observation is that participants often opted to *yield control* to the tool, likely as a means of reducing *cognitive load*. Rather than extensively structuring ChatGPT inputs or outputs, they leveraged the tool to generate substantial portions of the task. This behavior appeared to be effective in our context, where tasks were of *lower complexity*, and thus the risks associated with reduced user control were also lower.

However, control was not entirely absent. We identified naturally occurring strategies that helped participants *retain partial control* over the task without fully reasserting manual effort. Two notable prompting strategies, labeled as **C2** and **C3**, exemplify this. In **C3-$\mathbf{P_{IDEA}}$GPT$_{\mathbf{IMPL}}$**, participants provided the *test logic* and prompted ChatGPT to generate a fitting implementation. In **C2-GPT$_{\mathbf{IDEA}}$**$\boldsymbol{P_{IMPL}}$,

they asked ChatGPT for ideas of test cases and they then supplied the *implementation*. These approaches allowed users to offload parts of the task while anchoring the generation process in their own contributions, maintaining oversight and direction.

A further method to preserve control was the use of *iterative prompting* rather than one-shot generation. This strategy gave participants the ability to course-correct, reflect, and adjust ChatGPT outputs incrementally. Though probably more cognitively demanding than one-shot use, it served as a control mechanism that aligned well with some participants' cautious engagement with the tool.

While our tasks were intentionally limited in complexity, we can speculate that as *task complexity increases*, these lightweight control mechanisms may become insufficient. More complex problems not only introduce a higher cognitive burden for users, but may also push the boundaries of what ChatGPT can handle effectively. In such settings, we speculate that users, especially those with limited experience, would need to exert more deliberate and structured control over the tool to ensure correctness, coherence, and alignment with task goals. Lower experience can also lead to lower rigor for correctness. Even before the advent of generative AI technologies that promise to automate testing tasks, students struggled to grasp the value of testing (Pham et al., 2014). As generative AI automation becomes more widespread, being informed on why we test and what correctly done testing looks like becomes even more critical.

Recent findings on novice programming with generative AI by Prather et al. (2024) support this by suggesting that less experienced or lower self-efficacy students may struggle to engage productively with GenAI tools, sometimes developing misconceptions or an illusion of competence without realizing it.

4.5 A Broader Perspective and Implications

Although our findings are rooted in a low-experience sample working on relatively simple tasks, they contribute to a broader understanding of *how control is negotiated* in human-AI collaboration. We hypothesize that *experience* and *task complexity* are two key dimensions shaping the degree and nature of control required. While our participants leaned toward yielding control under manageable conditions, it remains an open question how this dynamic evolves as experience increases or task complexity rises.

Future research should examine how higher-experience users balance control and delegation, and whether similar strategies (e.g., C2, C3, iterative prompting) persist or evolve. Likewise, studies involving more complex, real-world programming tasks can test whether the informal control mechanisms observed here remain viable or require augmentation through information outside of users' personal experiences, such as training an AI agent for software testing.

By mapping out these relationships, we move closer to a fuller understanding of how control in generative AI-assisted workflows can support effective and responsible *software testing practices*.

In our study, students from a low-experience group tended to yield control to reduce cognitive load on lower-complexity tasks. Nevertheless, we observed naturally occurring strategies (C2-$\mathbf{GPT_{IDEA}}\boldsymbol{P_{IMPL}}$ and C3-$\mathbf{P_{IDEA}GPT_{IMPL}}$) and **iterative prompting** that helped preserve partial control over the process.

We propose a conceptual framework (Figure 6) that organizes control strategies across two key dimensions: user *experience* and *task complexity*. This model illustrates how different combinations of these factors may influence the selection and viability of prompting strategies in software testing. The low and high levels in this framework should be interpreted as relative conceptual dimensions rather than as precise thresholds; they are intended to illustrate how different conditions may shape interaction strategies.

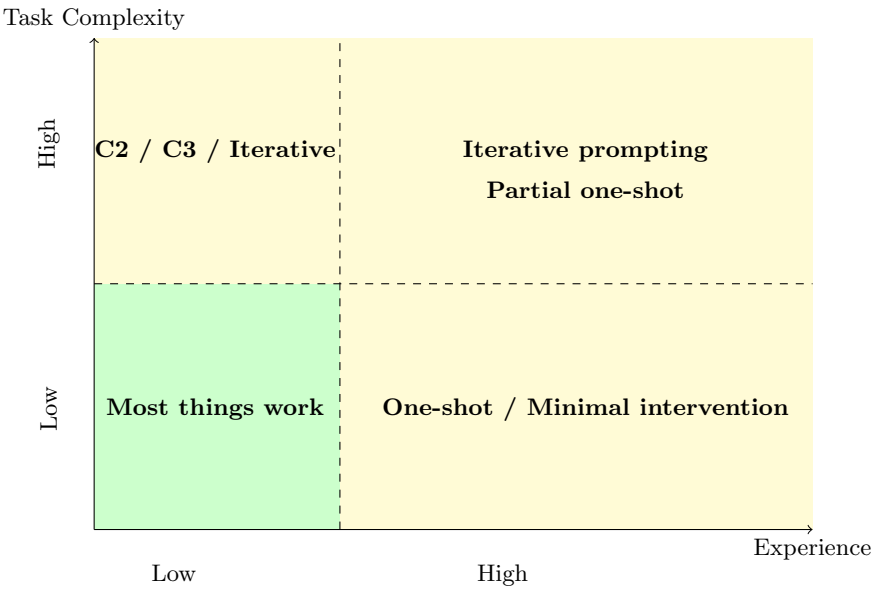

Fig. 6: Conceptual framework illustrating how experience and task complexity might shape the viability of different prompting strategies in AI-assisted software testing. The green quadrant is the scope of this study, while the yellow quadrants reflect our current hypothesis of what can be recommended.

This framework provides a foundation for understanding how users might balance cognitive effort and oversight when working with generative tools. While grounded in low-experience users and low-complexity tasks, it opens space for future research to explore how control strategies evolve across different levels of expertise and problem difficulty. Our findings provide an initial empirical basis for understanding how novice developers interact with generative AI in testing workflows. Ultimately, a more complete theory of generative AI-assisted *software testing* will require empirical studies with broader user groups and more complex, real-world scenarios.

Interpretation of the Framework:

- **Low Experience, Low Complexity**: Users often yield control, relying on one-shot prompting. C2 and C3 strategies, alongside iterative prompting, can still

emerge naturally to retain partial control over the process. However, lack of control related drawbacks are not observable due to low complexity.

- **Low Experience, High Complexity**: Users need to exercise more control to avoid negative outcomes. Iterative prompting and control-preserving strategies become more important to enable critical evaluation and adjustment of ChatGPT outputs.
- **High Experience, Low Complexity**: Users can effectively leverage one-shot prompting or minimal intervention strategies due to their stronger evaluation skills and domain expertise.
- **High Experience, High Complexity**: Even experienced users may shift toward iterative or reflective workflows to manage the increased likelihood of model errors or misalignment with task goals.

The main implication for researchers that follows from our work is that further investigation of the intricate relationship between developer experience, code complexity, prompting strategy, and GenAI experience is an important avenue. This relationship can be studied in the realm of engineering test code, but can likely also be studied more broadly when engineering production code.

An implication for developers is that awareness of the framework can encourage a more analytical approach to working with GenAI. Each strategy represents a different balance of control, cognitive effort, and reliance on the model, and understanding these trade-offs can help developers anticipate when closer oversight may be needed. Developers do not need to explicitly adopt one of the strategies. Rather, the strategies identified in this study provide a lens for understanding how users distribute control between themselves and generative AI during testing workflows. Being aware of these patterns may help developers reflect on their own interactions with generative AI tools and recognize potential trade-offs in different workflows.

The framework serves not only as an analytical lens but also as a means to encourage more reflective and informed AI-assisted testing practices.

5 Threats to Validity

We reflect on potential threats to the validity of our findings, dividing them into internal and external concerns. Where relevant, we outline measures taken to mitigate these risks.

5.1 Internal Validity

Two potential threats are related to *participant selection and task sequencing*. All participants were undergraduate students from a single institution who had completed a mandatory software testing course. While this ensured a minimal knowledge baseline, it limits the diversity of perspectives and experiences, introducing a selection bias. Additionally, participants completed two tasks in a fixed order during a single session. This raises the possibility of maturation effects, where experience gained in the first task influenced behavior in the second, although we did not counterbalance the task order to address this.

A second category of threats concerns *the influence of observation and data collection methods*. The think-aloud protocol, observation notes, and post-task interviews may have altered participant behavior. To mitigate this, we communicated to participants that the study focused on their usage patterns rather than performance outcomes. Additionally, the researcher conducting the observations had no involvement in teaching or assessment activities related to the participants. We also conducted preliminary pilot sessions to refine our procedures and familiarize both participants and researchers with the process. Moreover, we used multiple data sources, including video recordings, transcripts, and logs, to triangulate findings and reduce the impact of any one method. While the semi-structured interviews risk interviewer bias, a consistent interview guide helped maintain focus across sessions.

Finally, we acknowledge a potential threat related to *data analysis and measurement*. Since our analysis was based on qualitative open coding and thematic analysis, it involves an element of subjective judgment. To mitigate this bias, we performed interrater agreement checks.

5.2 External Validity

The generalizability of our results is limited by the *context and characteristics of our sample and tasks*. Our target audience was novice testers whose testing habits and workflows had not yet been shaped by extensive experience. To represent this group, we used students who had completed a mandatory undergraduate software testing course. These students shared a common theoretical foundation and some practical exposure from the course, but lacked substantial real-world experience, making them a suitable proxy for novice testers.

A related limitation concerns the *homogeneity and size of our participant sample*. All twelve participants had received similar prior instruction in software testing, which reduces diversity in testing strategies and may not reflect the broader population of software developers with varied backgrounds, professional experience, or exposure to different tools. The small sample size also means that the range of prompting behaviors and testing strategies we observed may not exhaust the full diversity present in practice. While a larger or more heterogeneous sample might alter the relative frequency of findings, we expect the categorical findings to remain relevant.

These participants worked on narrowly scoped unit testing assignments. This limits ecological validity, as professional testing often involves broader systems, messier codebases, team dynamics, and varying levels of experience among testers. These differences between our experimental setting and the complexity of real-world practice might make it difficult to extrapolate our findings directly to professional software development.

Another key limitation is *tool specificity*. Our findings are based on interactions with a single version of GPT-3.5. Although the behaviors we observed may extend to other generative AI tools, differences in model quality, interface design, or prompting affordances could influence prompting strategies and perceived usefulness. Similarly, we focused exclusively on unit testing. Other testing practices, such as integration, regression, or exploratory testing, may yield different patterns of engagement and utility when paired with generative AI.

Lastly, we note a concern related to *temporal validity*. As generative AI tools evolve rapidly, both in capabilities and user familiarity, the behaviors we observed may shift over time. However, by documenting the strategies and prompting styles in detail, we provide a snapshot that can serve as a baseline for future studies exploring how AI use in software testing matures. Additionally, the narrow scope of the assignments helps mitigate this concern, as the tasks were well within the current capabilities of AI. The AI's effective handling of these tasks, including generating relevant test ideas, indicates that participants' prompts were appropriately understood.

6 Related Work

Recent research on large language models (LLMs) in software engineering has primarily focused on two areas: evaluating their effectiveness in technical tasks and examining their use in educational settings. We discuss both topics in the subsequent subsections.

6.1 LLMs for Software Engineering Tasks

The performance and usability of ChatGPT is being considered for many use cases in software engineering. White et al. (2023) proposed a catalog of prompt engineering patterns to guide ChatGPT in tasks such as requirements elicitation and code refactoring. Their structured prompting strategies aim to improve the consistency, clarity, and maintainability of generated outputs.

Jalil et al. (2023) evaluated ChatGPT's performance on software testing questions, finding that it produced correct or partially correct answers 55.6% of the time, with explanations reaching similar accuracy. The study also showed that contextual prompting improved performance, although the model's self-reported confidence was not a reliable indicator of correctness.

For LLM applications in test generation, Jain and Le Goues (2025) introduced *TestForge*, an agentic framework that iteratively refines test suites using feedback from code execution and coverage reports. Starting from zero-shot outputs, the system dynamically updates test cases to improve performance and cost-efficiency. On the TestGenEval benchmark, TestForge achieved higher accuracy and coverage than both classical genetic programming tools and non-agentic LLM baselines, demonstrating the value of feedback-driven refinement in automated test generation.

Mock et al. (2024) explored generative AI use in test-driven development. Their initial findings report that generative AI can accelerate the process and reduce effort, while cautioning that use without active developer oversight may lead to brittle or misleading code, particularly for less experienced developers.

Compared to these studies, which primarily assess LLMs as automated tools or evaluate their output quality, our work focuses on how novice testers actually interact with LLMs during test generation. Rather than benchmarking accuracy or coverage, we investigate the strategies, behaviors, and decision-making processes that shape human–AI collaboration during unit testing.

6.2 Student Use of Generative AI

Other studies have explored the integration of generative AI tools into software engineering by studying students. Haldar et al. (2025) examined how postgraduate students used ChatGPT and Microsoft Copilot to generate testing artifacts. While students found the tools helpful for speeding up their work, they emphasized the continued need for manual oversight to ensure quality. Our study complements this finding by investigating how such oversight manifested, documenting the specific ways participants guided and corrected the AI and the varying degrees of human involvement observed during test creation.

Similarly, Mezzaro et al. (2024) observed that students who relied heavily on a GPT-based assistant within a gamified platform produced fewer effective test cases and demonstrated weaker learning outcomes. These results underscore the importance of guided use. Compared to our study, which involved students already trained in software testing, we did not observe a degradation in the quality of test suites when participants relied more heavily on generative AI. This contrast suggests that AI-based assistance may be a safer and more effective fit in contexts where users are practising established skills rather than still acquiring foundational testing knowledge.

Kazemitabaar et al. (2023) conducted a study with children aged 10 to 17 who used an AI code generator while learning Python and identified four coding approaches: AI Single Prompt, AI Step by Step, Hybrid, and Manual coding. These patterns resemble the strategies we identified in this study at the highest and lowest levels of reliance on GenAI: their AI Single Prompt approach parallels our C1 strategy and one-shot prompting pair, while the Manual coding approach also appeared directly in our data. A key contribution of our work is that we examined the space between heavy GenAI reliance and avoidance through a software testing lens, allowing us to articulate intermediate mitigation strategies and offer guidance for more informed GenAI use in testing contexts.

Rahe and Maalej (2025) conducted an in-depth study on how undergraduate programming students use ChatGPT during a programming exercise. Their findings reveal that many students, especially those with prior GenAI experience, gravitate toward using ChatGPT for direct code generation rather than for support or conceptual understanding. A common pattern observed was a cycle of requesting code, submitting the output, encountering errors, and re-prompting without critical reflection—a strategy that often proved ineffective. The study also highlights how overreliance on GenAI can lead to diminished autonomy and problem-solving agency, raising concerns for software engineering education.

Daun and Brings (2023) took a broader view, arguing that the rise of generative AI calls for changes to both teaching practices and curricula. They recommended shifting the focus from coding to design and critical thinking, and emphasized the need for supervised use of AI tools to avoid reinforcing misconceptions.

Choudhuri et al. (2024a) evaluated ChatGPT's role in supporting students with software engineering tasks through a controlled study. They found no improvement in productivity or self-efficacy, but significantly higher frustration among students using ChatGPT. The study identified recurring issues such as hallucinated responses

and incomplete assistance, which were linked to violations of Human AI interaction guidelines, highlighting the need for better alignment with educational goals.

In contrast to these studies, which focus either on productivity, learning outcomes, or general attitudes toward GenAI, our work provides a fine-grained behavioral analysis of how novice testers use LLMs specifically for unit test generation. We contribute new insights into the strategies students adopt (e.g., where ideas originate, how implementations are delegated), the prevalence of generated test content, and how these interactions shape the resulting test suites, thereby expanding the understanding of GenAI use from high-level outcomes to the concrete mechanics of human–AI collaboration during testing tasks.

7 Conclusion

This study examined how novice software developers interact with generative AI during unit test engineering tasks. Through an observational approach involving task recordings, the think-aloud method, and post-task interviews, we investigated three key questions: what strategies do students adopt when incorporating generative AI into unit testing workflows, how they formulate prompts, and what benefits and challenges emerge from such AI-assisted workflows.

Our findings reveal that participants employed a spectrum of strategies, ranging from full reliance on AI for both ideation and implementation (C1- $\mathbf{GPT_{IDEA}GPT_{IMPL}}$), to mostly manual workflows (C4- $\mathbf{P_{IDEA}}\boldsymbol{P_{IMPL}}$), with intermediate approaches (C2- $\mathbf{GPT_{IDEA}}\boldsymbol{P_{IMPL}}$ and C3- $\mathbf{P_{IDEA}GPT_{IMPL}}$) enabling varying degrees of user control. Prompting styles followed a similar spectrum, with participants opting for either one-shot generation of entire test suites or iterative prompting to guide ChatGPT incrementally. These strategies were not arbitrary: participants tended to match prompting style to their overall approach, participants who relied more heavily on ChatGPT for both ideation and implementation (C1- $\mathbf{GPT_{IDEA}GPT_{IMPL}}$) were more likely to favor one-shot prompting. At the same time, manual or control-preserving workflows leaned more on iterative engagement.

Students reported benefits, particularly in terms of time-saving and reduced cognitive load, suggesting that generative AI can streamline repetitive aspects of test creation and help shift attention to more analytical parts of the task. However, several drawbacks were also noted, including lack of trust, concerns regarding low quality of generated test cases, and a diminished sense of ownership. Notably, the most common errors were subtle assertion mismatches that required human oversight, highlighting that while ChatGPT can support ideation effectively, it falls short of fully automating high quality test implementation end-to-end.

In discussing these findings, we propose a conceptual framework linking user experience, task complexity, and control strategies. Our results suggest that low-experience users working on low-complexity tasks often cede control to ChatGPT, but still develop natural strategies (e.g., iterative prompting, partial delegation via C2- $\mathbf{GPT_{IDEA}}\boldsymbol{P_{IMPL}}$ and C3- $\mathbf{P_{IDEA}GPT_{IMPL}}$) that preserve agency. As task complexity and user expertise increase, we anticipate a shift toward more structured and deliberate control mechanisms.

This framework opens several promising directions for future research. Studies involving more experienced developers or more complex testing tasks could test whether the observed strategies generalize or evolve. Additionally, future work could investigate whether the prompting strategies and control dynamics observed here extend to other testing contexts, such as integration or non-functional testing, and explore support mechanisms that help students critically evaluate generated outputs, refine partially correct suggestions, and better balance control and automation.

In future work, we will also examine whether the conceptual framework identified in this study extends beyond unit test generation to other software engineering tasks, such as production code development. While our framework emerged from observing testing workflows, the dimensions of idea generation (human vs. GenAI) and implementation (human vs. GenAI) may also apply to broader programming activities. Investigating this possibility would help determine the generality of our framework.

Another promising direction is comparing our findings on ChatGPT with other more development focused generative AI tools (e.g., GitHub Copilot or Cursor), to examine whether different tooling ecosystems elicit distinct prompting behaviors, control strategies, or perceptions of usefulness. Such comparative studies would clarify whether the interaction patterns reported here reflect properties of conversational workflows specifically or generalize across other GenAI based development tools.

Such efforts may contribute to establishing best practices for responsible and effective AI-assisted software testing.

Appendix A Java Methods Used in the Experiment

Here we list the Java classes containing the methods used in the experiment.

```java
package delft;
public class InitialsUtils {
    private static boolean isEmpty(final CharSequence cs) {
        return cs == null || cs.length() == 0;
    }
    private static final String EMPTY = "";

    /**
     * Is the character a delimiter.
     *
     * @param ch          the character to check
     * @param delimiters  the delimiters
     * @return true if it is a delimiter
     */
    private static boolean isDelimiter(final char ch, final char[] delimiters) {
        if (delimiters == null) {
            return Character.isWhitespace(ch);
        }
        for (final char delimiter : delimiters) {
            if (ch == delimiter) {
                return true;
            }
        }
        return false;
    }

    /**
     * Extracts the initial characters from each word in the String.
     *
     * All first characters after the defined delimiters are returned as a new string.
     * Their case is not changed.
     *
     * If the delimiters array is null, then whitespace is used.
     * A {@code null} input String returns {@code null}.
     * An empty delimiter array returns an empty String.
     *
     * @param str         the String to extract initials from, may be null
     * @param delimiters  set of delimiter characters, null means whitespace
     * @return a String of initial characters, or {@code null} if input is null
     */
    public static String initials(final String str) {
        return initials(str, null);
    }
    public static String initials(final String str, final char... delimiters) {
        if (isEmpty(str)) {
            return str;
        }
        if (delimiters != null && delimiters.length == 0) {
            return EMPTY;
        }
        final int strLen = str.length();
        final char[] buf = new char[strLen / 2 + 1];
        int count = 0;
        boolean lastWasGap = true;
        for (int i = 0; i < strLen; i++) {
            final char ch = str.charAt(i);
            if (isDelimiter(ch, delimiters)) {
                lastWasGap = true;
            } else if (lastWasGap) {
                buf[count++] = ch;
                lastWasGap = false;
            }
        }
        return new String(buf, 0, count);
    }
}
```

Listing 5: The `initials` method used in the experiment.

```java
package delft;

import java.util.ArrayList;
import java.util.List;

public class SubstringsBetweenUtils {

    private static final String[] EMPTY_STRING_ARRAY = new String[0];

    private static boolean isEmpty(final CharSequence cs) {
        return cs == null || cs.length() == 0;
    }

    /**
     * Searches a String for substrings delimited by a start and end tag,
     * returning all matching substrings in an array.
     *
     * A {@code null} input String returns {@code null}.
     * A {@code null} open/close returns {@code null}.
     * An empty ("") open/close returns {@code null}.
     *
     * @param str   the String containing the substrings, null returns null
     * @param open  the String identifying the start delimiter
     * @param close the String identifying the end delimiter
     * @return an array of substrings, or {@code null} if no match
     */
    public static String[] substringsBetween(
            final String str, final String open, final String close) {

        if (str == null || isEmpty(open) || isEmpty(close)) {
            return null;
        }
        final int strLen = str.length();
        if (strLen == 0) {
            return EMPTY_STRING_ARRAY;
        }

        final int closeLen = close.length();
        final int openLen = open.length();
        final List<String> list = new ArrayList<>();

        int pos = 0;
        while (pos < strLen - closeLen) {
            int start = str.indexOf(open, pos);
            if (start < 0) break;

            start += openLen;
            final int end = str.indexOf(close, start);
            if (end < 0) break;

            list.add(str.substring(start, end));
            pos = end + closeLen;
        }

        if (list.isEmpty()) {
            return null;
        }
        return list.toArray(EMPTY_STRING_ARRAY);
    }
}
```

Listing 6: The `substringsBetween` method used in the experiment.

Declarations

Funding: This research is sponsored by the Swiss National Science Foundation (SNSF Grant 200021M 205146), the Dutch Science Foundation NWO through the Vici "TestShift" project (No. VI.C.182.032).

Ethical approval: This study was reviewed and approved by the Human Research Ethics Committee of Delft University of Technology.

Informed consent: Informed consent was obtained from all individual participants included in the study.

Author contributions: Andy Zaidman: Conceptualization, Supervision, Writing-review & editing, Funding acquisition.

Data availability statement: All data and analysis artifacts supporting the findings of this study are available at DOI [10.6084/m9.figshare.29429636.v1]. Personally identifiable data, such as raw interview transcripts and screen/audio recordings, are not publicly shared due to ethical and privacy considerations.

Conflict of interest: The authors declare that they have no conflict of interest.

Clinical trial number: Not applicable.

References

Al Nazi Z, Hossain MR, Al Mamun F (2025) Evaluation of open and closed-source LLMs for low-resource language with zero-shot, few-shot, and chain-of-thought prompting. Natural Language Processing Journal 10:100124

Aniche M, Hermans F, van Deursen A (2019) Pragmatic software testing education. In: Proceedings of the 50th ACM Technical Symposium on Computer Science Education, pp 414–420

Aniche M, Treude C, Zaidman A (2022) How developers engineer test cases: An observational study. IEEE Trans on Software Eng 48(12):4925–4946

Apache Software Foundation (2024) Apache commons lang. URL https://commons.apache.org/proper/commons-lang/, accessed: 2025-06-10

Ardic B, Zaidman A (2023) Hey teachers, teach those kids some software testing. In: 2023 IEEE/ACM 5th International Workshop on Software Engineering Education for the Next Generation (SEENG), IEEE, pp 9–16

Ardic B, Brandt C, Khatami A, Swillus M, Zaidman A (2025a) The qualitative factor in software testing: A systematic mapping study of qualitative methods. Journal of Systems and Software 227:112447, DOI https://doi.org/10.1016/j.jss.2025.112447

Ardic B, Le Dilavrec Q, Zaidman A (2025b) Data package for "how students use Generative AI for software testing: An observational study". DOI 10.6084/m9.figshare.29429636.v1

Athanasiou D, Nugroho A, Visser J, Zaidman A (2014) Test code quality and its relation to issue handling performance. IEEE Trans Software Eng 40(11):1100–1125, DOI 10.1109/TSE.2014.2342227

Camilleri D, Micallef M, Porter C (2022) Investigating cognitive workload during comprehension and application tasks in software testing. In: The 34th International Conference on Software Engineering and Knowledge Engineering (SEKE), pp 237–242

Choudhuri R, Liu D, Steinmacher I, Gerosa M, Sarma A (2024a) How far are we? the triumphs and trials of generative ai in learning software engineering. In: Proceedings of the International Conference on Software Engineering (ICSE), ACM, DOI 10.1145/3597503.3639201

Choudhuri R, Ramakrishnan A, Chatterjee A, Trinkenreich B, Steinmacher I, Gerosa M, Sarma A (2024b) Insights from the frontline: Genai utilization among software engineering students. arXiv preprint arXiv:241215624

Daun M, Brings J (2023) How chatgpt will change software engineering education. In: Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1, pp 110–116

Deljouyi A, Koohestani R, Izadi M, Zaidman A (2025) Leveraging large language models for enhancing the understandability of generated unit tests. In: Proceedings of the International Conference on Software Engineering (ICSE), IEEE, pp 1449–1461

Dilavrec QL, Zaidman A (2025) Hyperast: Incrementally mining large source code repositories. In: 22nd IEEE/ACM International Conference on Mining Software Repositories (MSR), IEEE, pp 461–464

El Haji K, Brandt CE, Zaidman A (2024) 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), ACM, pp 45–55, DOI 10.1145/3644032.3644443

Falessi D, Juristo N, Wohlin C, Turhan B, Münch J, Jedlitschka A, Oivo M (2018) Empirical software engineering experts on the use of students and professionals in experiments. Empirical Software Engineering 23(1):452–489

Fan Y, Tang L, Le H, Shen K, Tan S, Zhao Y, Shen Y, Li X, Gašević D (2025) Beware of metacognitive laziness: Effects of generative artificial intelligence on learning motivation, processes, and performance. British Journal of Educational Technology 56(2):489–530

Gisev N, Bell JS, Chen TF (2013) Interrater agreement and interrater reliability: key concepts, approaches, and applications. Research in Social and Administrative Pharmacy 9(3):330–338

Greiler M, Zaidman A, van Deursen A, Storey MD (2013) Strategies for avoiding text fixture smells during software evolution. In: Proceedings of the 10th Working Conference on Mining Software Repositories (MSR), IEEE, pp 387–396

Haldar S, Pierce M, Capretz LF (2025) Exploring the integration of generative ai tools in software testing education: A case study on ChatGPT and Copilot for preparatory testing artifacts in postgraduate learning. IEEE Access 13:46070–46090, DOI 10.1109/ACCESS.2025.3545882

Hassan AE, Lin D, Rajbahadur GK, Gallaba K, Côgo FR, Chen B, Zhang H, Thangarajah K, Oliva GA, Lin JJ, Abdullah WM, Jiang ZMJ (2024) Rethinking software engineering in the era of foundation models: A curated catalogue of challenges in the development of trustworthy fmware. In: d'Amorim M (ed) Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE), ACM, pp 294–305, DOI 10.1145/3663529.3663849

Jain K, Le Goues C (2025) Testforge: Feedback-driven, agentic test suite generation. arXiv preprint arXiv:250314713

Jalil S, Rafi S, LaToza TD, Moran K, Lam W (2023) ChatGPT and software testing education: Promises & perils. In: 2023 IEEE international conference on software testing, verification and validation workshops (ICSTW), IEEE, pp 4130–4137

Jia Y, Harman M (2010) An analysis and survey of the development of mutation testing. IEEE transactions on software engineering 37(5):649–678

Kazemitabaar M, Hou X, Henley A, Ericson BJ, Weintrop D, Grossman T (2023) How novices use LLM-based code generators to solve CS1 coding tasks in a self-paced learning environment. In: Proceedings of the 23rd Koli calling international conference on computing education research, pp 1–12

Le Dilavrec Q, Khelladi DE, Blouin A, Jézéquel JM (2023) Hyperdiff: Computing source code diffs at scale. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), ACM, pp 288–299, DOI 10.1145/3611643.3616312

Majdoub Y, Ben Charrada E (2024) Debugging with open-source large language models: An evaluation. In: Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), ACM, p 510–516, DOI 10.1145/3674805.3690758

Mattermost (2024) Mattermost - open source collaboration for developers. https://github.com/mattermost/mattermost, accessed: 2025-06-13

McKnight PE, Najab J (2010) Mann-Whitney U test. The Corsini encyclopedia of psychology pp 1–1

Meszaros G (2007) xUnit Test Patterns: Refactoring Test Code. Addison-Wesley

Mezzaro S, Gambi A, Fraser G (2024) An empirical study on how large language models impact software testing learning. In: Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering, ACM, pp 555–564

Mock M, Melegati J, Russo B (2024) Generative ai for test driven development: Preliminary results. In: International Conference on Agile Software Development, Springer, pp 24–32

Molina F, Gorla A, d'Amorim M (2025) Test oracle automation in the era of LLMs. ACM Trans Softw Eng Methodol 34(5), DOI 10.1145/3715107

Moonen L, van Deursen A, Zaidman A, Bruntink M (2008) On the interplay between software testing and evolution and its effect on program comprehension. In: Mens T, Demeyer S (eds) Software Evolution, Springer, pp 173–202

Peruma A, Almalki K, Newman CD, Mkaouer MW, Ouni A, Palomba F (2020) 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), ACM, pp 1650–1654

Petrović G, Ivanković M, Fraser G, Just R (2021) Does mutation testing improve testing practices? In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), IEEE, pp 910–921, DOI 10.1109/ICSE43902.2021.00087

Pham R, Kiesling S, Liskin O, Singer L, Schneider K (2014) Enablers, inhibitors, and perceptions of testing in novice software teams. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp 30–40

Prather J, Reeves BN, Leinonen J, MacNeil S, Randrianasolo AS, Becker BA, Kimmel B, Wright J, Briggs B (2024) The widening gap: The benefits and harms of generative ai for novice programmers. In: Proceedings of the 2024 ACM Conference on

International Computing Education Research — Volume 1, ACM, pp 469–486
Przymus P, Fejzer M, Narundefinedbski J, Stencel K (2024) How i learned to stop worrying and love chatgpt. In: Proceedings of the 21st International Conference on Mining Software Repositories, Association for Computing Machinery, New York, NY, USA, MSR '24, DOI 10.1145/3643991.3645073
Rahe C, Maalej W (2025) How do programming students use generative AI? Proc ACM Softw Eng 2(FSE), DOI 10.1145/3715762
Russo D, Baltes S, van Berkel N, Avgeriou P, Calefato F, Cabrero-Daniel B, Catolino G, Cito J, Ernst N, Fritz T, et al. (2024) Generative AI in software engineering must be human-centered: The Copenhagen manifesto. Journal of Systems and Software 216:112115
Salman I, Misirli AT, Juristo N (2015) Are students representatives of professionals in software engineering experiments? In: 2015 IEEE/ACM 37th IEEE international conference on software engineering, IEEE, vol 1, pp 666–676
Schecter A, Richardson B (2025) How the role of generative AI shapes perceptions of value in human-AI collaborative work. In: Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems, ACM, DOI 10.1145/3706598.3713946
Spadini D, Palomba F, Zaidman A, Bruntink M, Bacchelli A (2018) On the relation of test smells to software code quality. In: International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp 1–12
Treude C, Storey MD (2025) Generative AI and empirical software engineering: A paradigm shift. In: 2nd IEEE/ACM International Conference on AI-powered Software (AIware), IEEE, pp 233–239
Trinkenreich B, Calefato F, Hanssen G, Blincoe K, Kalinowski M, Pezzè M, Tell P, Storey MA (2025) Get on the train or be left on the station: Using LLMs for software engineering research. In: 33rd ACM International Conference on the Foundations of Software Engineering (FSECompanion'25), pp 0–0
Wang J, Huang Y, Chen C, Liu Z, Wang S, Wang Q (2024) Software testing with large language models: Survey, landscape, and vision. IEEE Transactions on Software Engineering 50(4):911–936
Wasi AT, Islam MR, Islam R (2024) LLMs as writing assistants: Exploring perspectives on sense of ownership and reasoning. In: Proceedings of the Third Workshop on Intelligent and Interactive Writing Assistants, pp 38–42
Weber T, Brandmaier M, Schmidt A, Mayer S (2024) Significant productivity gains through programming with large language models. Proc ACM Hum-Comput Interact 8(EICS), DOI 10.1145/3661145
White J, Fu Q, Hays S, Sandborn M, Olea C, Gilbert H, Elnashar A, Spencer-Smith J, Schmidt DC (2023) A prompt pattern catalog to enhance prompt engineering with chatgpt. arXiv preprint arXiv:230211382
Williams M, Moser T (2019) The art of coding and thematic exploration in qualitative research. International management review 15(1):45–55
Willig C, Rogers WS (2017) The SAGE handbook of qualitative research in psychology. Sage, London, UK
Zhu H, Hall PAV, May JHR (1997) Software unit test coverage and adequacy. ACM Comput Surv 29(4):366—427, DOI 10.1145/267580.267590

Ziegler A, Kalliamvakou E, Li XA, Rice A, Rifkin D, Simister S, Sittampalam G, Aftandilian E (2024) Measuring GitHub Copilot's impact on productivity. Commun ACM 67(3):54–63, DOI 10.1145/3633453