



# An Empirical Study on How Large Language Models Impact Software Testing Learning

Simone Mezzaro  
simone.mezzaro@mail.polimi.it  
Politecnico di Milano  
Italy

Alessio Gambi  
alessio.gambi@ait.ac.at  
Austrian Institute of Technology  
Austria

Gordon Fraser  
gordon.fraser@uni-passau.de  
University of Passau  
Germany

## ABSTRACT

Software testing is a challenging topic in software engineering education and requires creative approaches to engage learners. For example, the Code Defenders game has students compete over a Java class under test by writing effective tests and mutants. While such gamified approaches deal with problems of motivation and engagement, students may nevertheless require help to put testing concepts into practice. The recent widespread diffusion of Generative AI and Large Language Models raises the question of whether and how these disruptive technologies could address this problem, for example, by providing explanations of unclear topics and guidance for writing tests. However, such technologies might also be misused or produce inaccurate answers, which would negatively impact learning. To shed more light on this situation, we conducted the first empirical study investigating how students learn and practice new software testing concepts in the context of the Code Defenders testing game, supported by a smart assistant based on a widely known, commercial Large Language Model. Our study shows that students had unrealistic expectations about the smart assistant, “blindly” trusting any output it generated, and often trying to use it to obtain solutions for testing exercises directly. Consequently, students who resorted to the smart assistant more often were less effective and efficient than those who did not. For instance, they wrote 8.6% fewer tests, and their tests were not useful in 78.0% of the cases. We conclude that giving unrestricted and unguided access to Large Language Models might generally impair learning. Thus, we believe our study helps to raise awareness about the implications of using Generative AI and Large Language Models in Computer Science Education and provides guidance towards developing better and smarter learning tools.

## CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Social and professional topics** → **Software engineering education**.

## KEYWORDS

Smart Learning Assistant, Generative AI, Computer Science Education, ChatGPT



This work is licensed under a Creative Commons Attribution International 4.0 License.

EASE 2024, June 18–21, 2024, Salerno, Italy

© 2024 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-1701-7/24/06

<https://doi.org/10.1145/3661167.3661273>

## ACM Reference Format:

Simone Mezzaro, Alessio Gambi, and Gordon Fraser. 2024. An Empirical Study on How Large Language Models Impact Software Testing Learning. In *28th International Conference on Evaluation and Assessment in Software Engineering (EASE 2024)*, June 18–21, 2024, Salerno, Italy. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/3661167.3661273>

## 1 INTRODUCTION

Software testing is one of the most challenging software engineering topics to teach. Consequently, educators often resort to creative approaches to improve the learning environments for students. As a prominent example, the Code Defenders testing game [3, 13] aims to engage students by turning mutation testing into a competitive game. To turn this engagement into effective practice, however, students may require assistance, for example, to understand testing technologies or the code under test.

The recent advent of Large Language Models (LLMs) may provide an opportunity to address this problem. The introduction of OpenAI’s ChatGPT in 2022 [10] made the power of Large Language Models easily accessible to the general public thanks to its user-friendly interface and simplified interaction procedure employing textual chats. Such tools can support learners and educators by providing immediate explanations on unclear topics and generating exercises and other lecture materials, possibly tailor-made to students’ needs [12]. In particular, recent work [1] suggests the potential of LLMs in Software Engineering Education. For instance, ChatGPT can generate programming exercises [15], answer questions on fundamental software testing topics [7], create failure-inducing test cases [9] and write unit tests [14].

However, the unrestricted and unsupervised usage of LLMs and their possible hallucinations [6] might affect learning negatively. For instance, students might misuse LLMs to cheat, i.e., generate ready-made solutions for homework and assignments. This, in turn, might prevent students from putting effort into solving these problems, invalidate established evaluation methods, and undermine students’ reasoning capabilities and problem-solving skills.

We, therefore, seek to investigate whether and how GPT can impact Software Testing Education by integrating it into Code Defenders [13], an open-source web platform for teaching and practising Software Testing. As illustrated in Figure 1, we extended Code Defenders with a *smart assistant* called AI Defenders that leverages OpenAI’s chat completion API<sup>1</sup> and GPT-3.5 Turbo model<sup>2</sup> to answer student questions about generic software testing concepts and specific questions about the games they are currently playing.

<sup>1</sup><https://platform.openai.com/docs/api-reference/chat>

<sup>2</sup><https://platform.openai.com/docs/models/gpt-3-5-turbo>

Code Defenders Multiplayer player

Game #108 Melee Player 54min Scoreboard Timeline Gradle Export Feedback Editor Mode: default Chat

Smart Assistant Remaining Questions: 5 Previous Questions

Can you please write a simple test that checks the output of the toString() method for a default Options object?

Include code example in the answer Submit

Hide Smart Assistant

Create a mutant here Enemy Coverage Reset Attack

Options	OptionGroup	Option
1		import java.util.ArrayList;
2		import java.util.Collection;
3		import java.util.Collections;
4		import java.util.HashSet;
5		import java.util.LinkedHashMap;
6		import java.util.List;
7		import java.util.Map;

Write a new JUnit test here Defend

```

1 import java.util.ArrayList;
2 import java.util.Collection;
3 import java.util.Collections;
4 import java.util.HashSet;
5 import java.util.LinkedHashMap;
6 import java.util.List;
7 import java.util.Map;
8
9 import org.junit.Test;

```

Figure 1: User interface of a Code Defenders Melee game with the smart assistant AI Defenders enabled

Using AI Defenders, we aim to shed light on students' expectations about LLMs and how they use them. Moreover, we seek to understand whether and how GPT effectively improves students' learning experience. With this purpose in mind, we conducted a mixed-method empirical study in the context of an academic course on Software Testing at the University of Passau (Germany). Compared to existing empirical studies, which focused on programmers' perceptions and behaviour when using Large Language Models for code generation [17], our study focuses on students and how GPT could support learning and practising software testing.

Comparing students playing Code Defenders games with and without the smart assistant, we observe that providing them unrestricted access to GPT mediated by the smart assistant was often not beneficial. In line with our direct experience in class, we find that many students used the tool in the hope of generating ready-to-use answers (i.e., cheating). Additionally, students did not realise when their questions lacked important details that would have enabled GPT to generate appropriate answers. In most cases, students did not attempt to fix wrong code generated by GPT.

We therefore argue that the unsupervised use of GPT might impair learning, especially for students who overestimate its capabilities and cannot exploit it effectively to solve testing tasks, such as generating unit tests featuring mock objects. Therefore, we conjecture that many of the issues we discovered could be solved by guiding students toward the correct use of the tool, for example, by

warning them about the possible inaccuracy of code generated by GPT, modulating the type of requests they can make, and actively making suggestions to direct them toward the solution of a problem or effectively learning the underlying testing concepts.

## 2 CODE DEFENDERS

Code Defenders<sup>3</sup> [8] is a web-based platform that borrows concepts from mutation testing [11] and uses *gamification* to support learning and practice of Software Testing [8, 13]. As previous studies illustrate (e.g., [3]), Code Defenders enhances students' engagement and performance while practising software testing. Consequently, it has been successfully integrated as a learning tool in software testing courses at various universities.

Code Defenders games revolve around Java classes, the unit of code under test or *CUT*, that are chosen at game creation. Players (e.g., students) compete by creating tests and mutants for the CUT. In particular, *attackers* create mutants by injecting subtle faults in the CUT, whereas *defenders* write unit tests to detect these mutants, i.e., *killing* them. As illustrated in Figure 1, Code Defenders provides an intuitive graphical user interface (GUI) with advanced functionalities (e.g., code completion) that allow players to mutate the CUT or generate unit tests using the reference JUnit library<sup>4</sup>.

<sup>3</sup><https://code-defenders.org/>

<sup>4</sup><https://junit.org/junit5/>

Games have a fixed *duration*, which is configured when they are created. During a game, attackers gain points based on the number of tests that cover but do not kill their mutants; thus, mutants that are exercised by many tests without being killed score more points. In contrast, defenders receive points based on the number of mutants their tests kill; thus, more effective tests that have the ability to spot several faults have a higher value.

Attackers may create *equivalent* mutants that cannot be detected by any tests. Defenders can claim mutants as equivalent, which initiates equivalence duels: the attacker who created the supposedly equivalent mutant must provide a test that can kill it, thus proving that the mutation is not equivalent. If the attacker succeeds, they win the duel and the mutant retains its points; otherwise, the defender that triggered the duel gains one extra point.

Code Defenders allows players to practice independently by solving *puzzles* of increasing complexity or to compete against other players in teams or individually. In *battleground* games, players belong to either the attackers' or defenders' teams. Consequently, players can only mutate the CUT or write tests in these games, but not both. Each player's activity is completely visible to the other members of the same team and contributes to the overall team's score. The team that collectively scores more points wins the game. In *melee* games, each player competes against all the other players. Thus, there are no teams in this game, and each player can create mutants and tests for the CUT. Figure 1 shows Code Defenders's GUI for melee games below the interface of our novel smart assistant AI Defenders.

There is a configurable difficulty level: In *easy* games, defenders can inspect the code of mutants and compare it against the original one (i.e., code diff). Attackers, instead, can see the source code of all the submitted test cases and the CUT's line they exercise. In contrast, in *hard* games, attackers can only see the lines covered by the opponents' test cases, i.e., they cannot see the submitted test cases' source code. Likewise, defenders can only see the position of injected faults inside the CUT.

Code Defenders collects the following metrics about players' performance during a game:

- **Total Points:** the number of points scored.
- **Valid Tests** and **Valid Mutants:** respectively, the number of compiling tests and mutants.
- **Failing Tests** and **Failing Mutants:** respectively, the number of tests that either did not compile or failed on the original CUT and the number of mutants that did not compile.
- **Useful Tests:** the number of valid tests that killed at least one mutant.
- **Useful Mutants:** the number of valid mutants that survived at least one test that covers them.

For our study, we observed these metrics to understand whether and how using AI Defenders improved students' performance in the game or impaired them.

### 3 AI DEFENDERS IN A NUTSHELL

To learn how students interact with GPT in the context of Code Defenders, we developed AI Defenders, a smart assistant that captures

the questions students ask during the games, sends them to OpenAI's Chat API, parses the corresponding responses, and shows them back to the students in a structured way (see Figure 3).

Figure 2 illustrates how we integrated Code Defenders and GPT by means of AI Defenders. Code Defenders is a layered web application; therefore, we distributed AI Defenders's logic across its *frontend* and *backend* layers.

We extended the frontend of Code Defenders, which normally shows the CUT, the mutants and tests visible from the player, and the various game controls by adding the new *Smart Assistant* widget. As illustrated in Figure 1, this widget consists of a text box where students can type their questions (on the left) and a component listing previously asked questions along with the corresponding responses generated by GPT (on the right). Figure 3 shows a detailed example of a question and the corresponding answer we collected during our empirical study.

AI Defenders allows students to ask open-ended questions and does not enforce any constraints on the questions besides limiting their size to 1500 words. Since we expected questions might be related to existing tests or mutants, we implemented a reference mechanism that allows students to *tag* those game artefacts. For instance, a student might ask the smart assistant "*How to modify @test100 to cover the method mutated by @mutant25*". This mechanism allows students to contextualise their questions onto the games by easily including code snippets implementing specific tests or mutants. Noticeably, to prevent students from accessing information about opponents' tests and mutants that should not be disclosed using the assistant, AI Defenders includes only the information that is currently available to students via the original user interface of Code Defenders. Therefore, when playing games in easy mode (see Section 2), students can reference their own tests and mutants, tests and mutants of their team members, as well as tests and mutants of their opponents. In this case, AI Defenders includes the corresponding tests and mutants' code *verbatim* in the prompt. In contrast, while playing games in hard mode, students can reference their tests and mutants and only the locations of the opponents' mutants. Consequently, opponents' tests cannot be referenced as they are not visible to the students.

Finally, the Smart Assistant widget allows students to decide whether code examples should be included in the responses generated by GPT. This can be achieved by toggling the appropriate switch (see Figure 1) and allowed us to assess how much students value code examples in the responses during the empirical study.

As Figure 3 illustrates, responses are divided into three sections: a short *Answer*, a longer *Explanation*, and an optional *Code Example*. We designed this structure so that we could retrieve students' feedback for each part of the response and understand which ones they consider more relevant. Additionally, students can give immediate feedback upon receiving a response by selecting whether the answer was useful or not. However, AI Defenders does not enforce by any means providing such feedback.

To bridge the communication between the Smart Assistant and GPT, we expanded Code Defenders Games API with the new AI Defenders REST API and deployed it on Code Defenders *Backend*. Games API allows players to retrieve information about the state of the games, submit new tests or mutants, and start equivalence duels by claiming a mutant as equivalent. The AI Defenders API instead

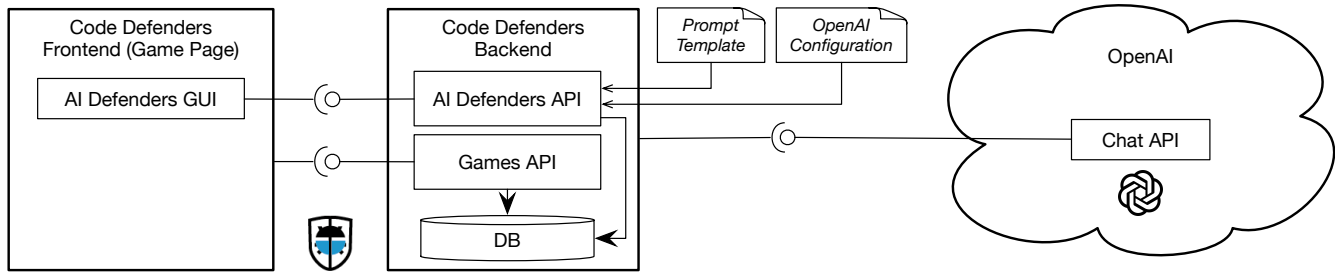



Figure 2: Overview of AI Defenders's integration with Code Defenders

Smart Assistant  Remaining Questions: 4

**Question:**  
Can you please write a simple test that checks the output of the toString() method for a default Options object?

**Answer:**  
Yes, we can write a simple test to check the output of the toString() method for a default Options object.

**Explanation:**  
We can create a default Options object, call the toString() method, and compare the output with the expected result.

**Code:**  
Options options = new Options();  
String expected = "[ Options: [ short {} ] [ long {} ]";  
String actual = options.toString();  
assertEquals(expected, actual);

Was the answer useful? Yes No New Question

Figure 3: Example of a GPT response from the assistant

receives students' questions and combines them with information about tests and mutants retrieved from the Code Defenders Database to form the *system prompt*, the actual request sent to OpenAI's chat completion API. AI Defenders generates the system prompt dynamically, starting from a predefined *prompt template* that contains additional information and instructions useful to generate the response. For instance, the system prompt includes the source code of the CUT and instructions describing how to structure the response. For our study, we leveraged established prompt engineering techniques [18] and trial-and-error to define a system prompt template that results in prompts similar to the one in Figure 4. Future research will investigate possible refinements and improvements of this system prompt template based on students' feedback.

According to this template, the system prompt instructs GPT to produce responses organised in sections following a predefined JSON scheme. Specifically, responses shall contain two mandatory parts: a one-sentence answer and a hundred-word explanation. Additionally, the responses could contain a code example if the students explicitly requested it. Although OpenAI's chat completion API allows including previous messages within new requests akin to an *interactive session* to build a context, we decided to always

You are given the Java class between "" "".

```

"""
import java.util.ArrayList;
[...]
public class Options {
    [...]
}
"""

```

Finally, you are given some tests for the Java class:  
@test100 has code:

```

[...]
public class TestOptions {
    @Test(timeout = 4000)
    public void test() throws Throwable {
        [...]
    }
}

```

Reply to the following question by providing

- a short answer using only one sentence
- an explanation of the answer using at most 100 words
- a code example

Provide the reply in JSON format with the following keys:  
answer, explanation, and code.

Figure 4: A system prompt generated by AI Defenders during our empirical study. The prompt contains CUT's source code, the original question asked by the student (omitted in the figure), and the source code of the tagged test.

and only submit the system prompt corresponding to the current question. The main reason behind our choice is to avoid potentially misleading dependencies between consecutive questions. However, we acknowledge the importance of building a proper context around students' requests; thus, we plan to investigate this issue more deeply in future research.

Finally, to help educators configure AI Defenders and monitor its usage, we developed a new administrative interface that Figure 5 depicts. This interface, which complements the existing Code Defenders administrative interface, consists of: (1) Submitted Questions. This widget shows the total number of questions submitted

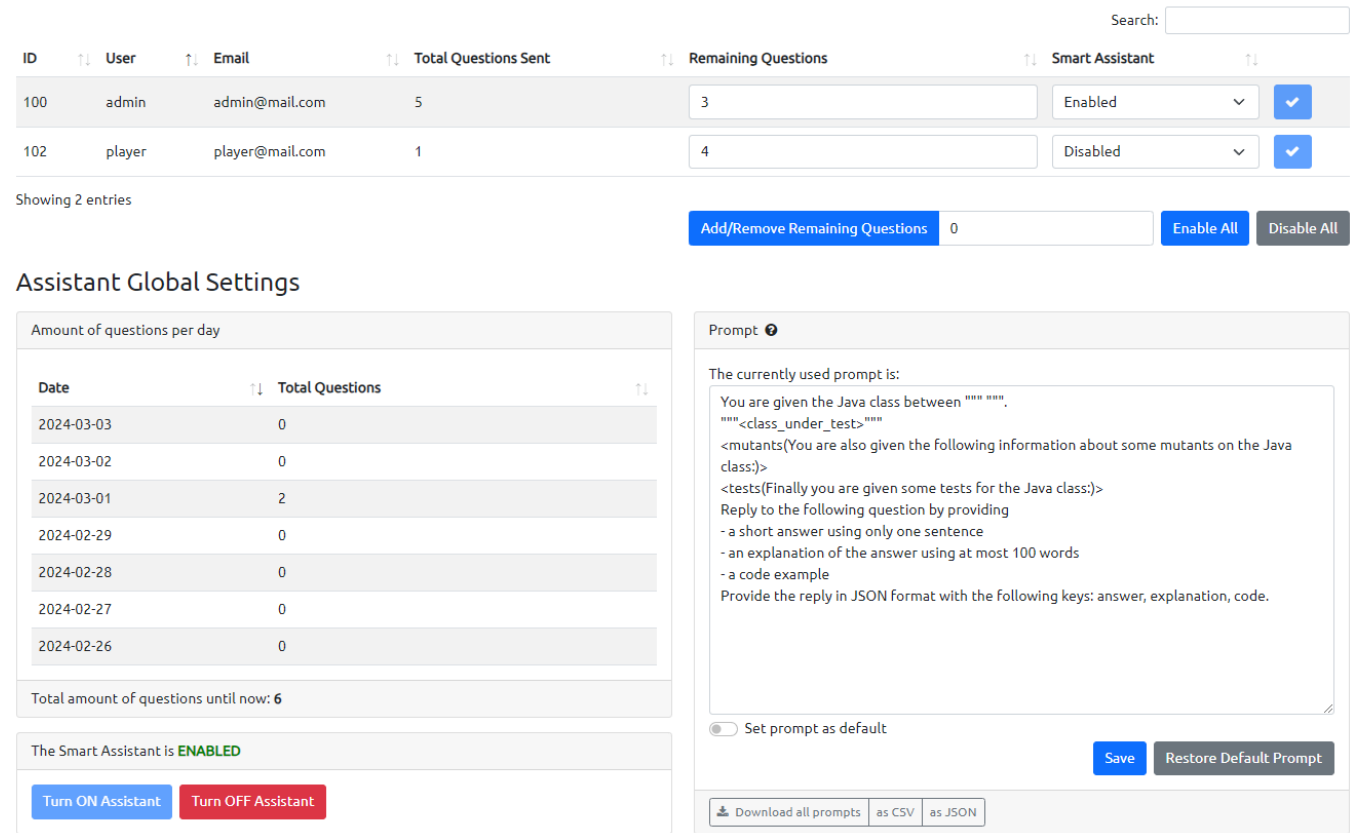


Figure 5: AI Defenders's administrative interface

by each student, allows granting students additional requests, and allows administrators to enable or disable the assistant for each student. This widget can limit the assistant's usage by capping the maximum number of questions each student can ask, thus avoiding excessive requests flooding OpenAI's API (and a high bill to the system administrators). (2) Assistant Global Settings. On one side (left), this widget allows administrators to track OpenAI's API usage per day or overall, and to enable or disable the assistant for all active games. On the other side (right), this widget allows administrators to update the system prompt template.

## 4 EXPERIMENTAL SETUP

We conducted an experiment using AI Defenders to collect qualitative and quantitative data about whether and how the tool impacts students' behaviour and performance. We analysed the gathered data to understand students' expectations of the assistant and their ability to leverage its responses to improve their learning and practice of Software Testing.

### 4.1 Research Questions

We seek answers to the following main research questions:

*RQ1: What are students' expectations about the GPT-based smart assistant?* Answering this question is important to understand

whether students are aware of the tool's effective capabilities and main limitations. To answer this research question, we analysed the student requests submitted to the smart assistant during the experiment. We also investigated how the students used the smart assistant to play Code Defender games featuring new testing topics.

*RQ2: To what extent do students find GPT's responses correct and useful?* Answering this question complements RQ1's findings by assessing how much students trust the responses generated by the smart assistant and whether they can effectively use those responses to improve learning and practising testing. To answer this question, we ran a survey at the end of the experiment and collected students' feedback on the quality and usefulness of the assistant responses. We also tracked whether students could exploit GPT responses to their advantage, for instance, by generating valid test cases.

*RQ3: Does students' performance improve using the smart assistant in a learning context?* We aim to understand the utility of using a LLM-based tool for learning and practising software testing. Answering this question is paramount to gaining such understanding. To answer this research question, we used the various performance metrics gathered by Code Defenders (see Section 2) to assess whether students produced better tests and mutants with GPT support or without it.

## 4.2 Design of the Empirical Study

We conducted our empirical study in the context of the software testing course taught at the University of Passau as an optional session at the end of the semester. The session took place in regular class settings under the supervision of trained teaching assistants. Students' performance was not graded, but participants were awarded bonus points on their final grade based on attendance.

In total, 39 students participated in the experiment; however, two of them left before its completion. Therefore, for our analysis, we consider only the observations collected from the 37 students who attended the whole session. All participants are Master's students enrolled in the Computer Science program. They are familiar with Code Defenders, as they played several games throughout the semester, i.e., before our study took place. However, since AI Defenders was not available in earlier lectures, it was briefly introduced to the students at the beginning of the experiment.

The experiment consisted of playing two consecutive rounds of melee games (see Section 2), each lasting about 30 minutes, in which only half of the students had access to the Smart Assistant. We used different classes under test to reduce learning effects between consecutive rounds. Specifically, we selected the *Options* class from *Apache Commons CLI* in Round 1, whereas we selected the *Document* class from *Apache Lucene* in Round 2. Those classes have comparable complexity, are commonly available on Code Defenders, and have been used in previous experiments [3].

*Options* and *Document* require dependencies to be correctly instantiated. In Code Defenders, those dependencies are replaced by abstract interfaces so that students can practice *mocking* [4]. Notably, this was the first time students had to use mocking, which allowed us to observe how students used the assistant to learn and put into practice an unfamiliar topic.

In each round, we randomly allocated students to games so that each game would roughly have the same size (4-5 players). We used melee games so that we could observe their behaviour while playing both the attacker and defender roles. We chose the hard level because the students were already familiar with Code Defenders. We enabled the smart assistant for only half of the games; thus, only students competing in those games could submit questions and receive responses generated by the GPT model. We used the other half of the games as a baseline for comparison.

By the end of the experiment, each student had played twice: once with the smart assistant enabled and once without it. However, the order in which they accessed the smart assistant was randomised. In Round 1, we ran 9 games, 4 of which had the assistant enabled; thus, a total of 18 students received support via the assistant. In Round 2, we also ran 9 games. This time, however, 5 games enabled the smart assistant and 19 students could use it. We ensured that students who got access to the smart assistant in Round 1 played without it in Round 2, and vice versa. We compared each student's performance when playing with and without the assistant. Table 1 summarises the setup for the two rounds.

Due to budget constraints, we set the maximum number of questions available to each student per game to 20. We believed this limit could prevent anomalous assistant usage but was conservative enough not to restrict students. Noticeably, after the study, we observed that none of the students reached this limit.

Finally, at the end of the session, we invited students to complete an optional survey. The survey aimed to assess to which degree the smart assistant met students' expectations and whether the usefulness and correctness of the smart assistant responses were satisfactory. Students were also free to provide additional feedback and suggest possible improvements for the smart assistant. In total, 33 (89%) students completed the survey.

## 4.3 Threats to Validity

Specific decisions we made while designing our tool and setting up our experiment may threaten the generalisability of this study.

First, adopting a particular LLM might cast doubt on the reliability of our observations. To address this concern, we picked the GPT-3.5 model by OpenAI as, at the time of writing, the improved GPT-4.0 was not publicly accessible. While different models could respond with a different degree of correctness, we opted for a popular LLM, which can be considered a good representative of the most diffused LLMs among students. Moreover, we focused on students' expectations and interactions, i.e., their questions and how they used responses, rather than studying the specific model's performance by analysing the content and correctness of its responses.

Implementation and configuration choices we made for AI Defenders threaten internal validity as they influence students' behaviour. For example, we implemented a simple interface for the smart assistant to provide students with an unconstrained and unguided environment and did not warn them that the tool might produce inaccurate (or plain wrong) responses. Another crucial element in our experiment is the system prompt. While we suspect the prompt's constraints on the responses' length might have affected some explanations returned by the model, we notice that most of the questions asked for (unrestricted) code examples.

Another threat to the validity of our conclusion comes from the rather small empirical study. Due to time and budget constraints, we could run only two rounds with relatively few students. Consequently, the achieved results might not be representative for more and longer sessions using different classes. To reduce this threat, we made each student compete with and without the assistant using different classes under test each time.

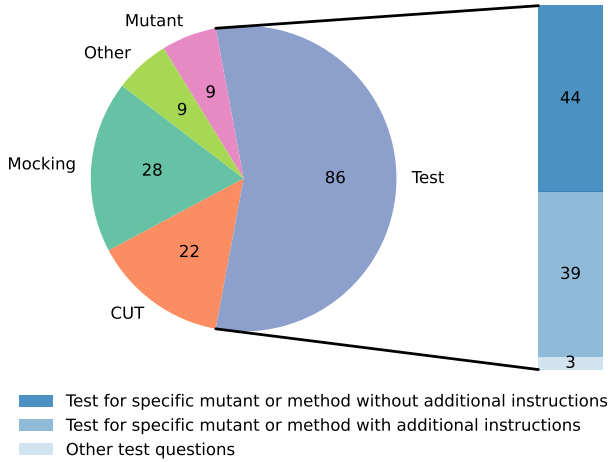
## 5 RESULTS

We collected all questions submitted by the students to AI Defenders during the two rounds of Code Defenders, together with the corresponding responses generated by the GPT model. This resulted in a total of 199 questions; however, we had to discard 27 questions because they were unrelated to the game or sent multiple times (by the same student). From the remaining 172 *valid questions*, we removed further 18 questions that, although related to the game and thus valid, were not written in English. We refer to the remaining 154 questions as *English questions*. Besides student questions and model responses, we recorded various metrics to assess students' performance, including metrics about students' activity (e.g., number of submitted tests and mutants), quality of submissions (e.g., number of valid, invalid, and useful tests and mutants), and performance (e.g., scored points).



**Table 1: Experimental setup for the two rounds of Code Defenders**

	Java class	Games with assistant	Games without assistant	Students with assistant	Students without assistant
Round 1	Options	4	5	18	19
Round 2	Document	5	4	19	18

**Figure 6: Classification of English questions.**

### 5.1 RQ1: What are students' expectations about the GPT-based smart assistant?

As the first step to understanding students' expectations about AI Defenders, we analysed what types of questions they asked. For this reason, one of the authors *manually* inspected all the English questions and classified them into one of the following five topics:

**Mocking.** Generic questions about mocking and how to use the mocking library supported by Code Defenders (i.e., Mockito) and focused questions on how to mock specific elements of the CUT or its dependencies.

**CUT.** Questions about the code under test, for example, to understand the behaviour of a method.

**Test.** Questions about tests, for instance, to explain a test or create a test targeting specific methods, mutants, or lines of code.

**Mutant.** Questions about mutants, for instance, to explain a mutation or create mutants of a CUT's methods or lines of code.

**Other.** Questions that do not clearly belong to any of the previous classes.

As illustrated in Figure 6, the majority of submitted questions regard tests (55.8%) and mocking (18.2%), while only a small fraction of the questions is related to the CUT (14.3%), mutants (5.8%), and other topics (5.8%). Moreover, students explicitly requested additional code examples in 68.2% of the English questions.

These results match our expectations that students would ask many questions on tests and mocking since they had to write tests using an unfamiliar mocking library. Our conclusion is that students

assumed the smart assistant could provide valuable explanations and generate code useful for applying mocking in practice. In contrast, students did not rely on the smart assistant to perform other tasks, like mutating the CUT, which they had already practised and felt confident about.

Focusing on the questions regarding tests (Figure 6, on the right), we can observe that in 96.5% of them (corresponding to 53.9% of the English questions), students asked the smart assistant to generate a (ready-to-use) test targeting a specific mutant or CUT's method. Only 47.0% of those questions contain instructions to help the smart assistant complete the task. For instance, the questions contain details about the objects to mock or suggestions for creating the assertions. The remaining 53.0% of the questions merely ask the smart assistant to produce the test code. An example of questions that belong to the latter group reads as follows:

*Give me a test for the toString method of the Options class*

An example question from the former group is the following:

*Write a test that tests that the removeField method only removes the first field with a given name, not all of them. There are no concrete implementations of IndexableField and BytesRef available, so these need to be mocked with Mockito. Do not check if the function was called with Mockito, but check if the field has really been added to and removed from the Document. Also, be aware that Mockito and Assert are already imported, so no function call prefixes for these libraries are needed.*

The question is more elaborate in this case, as the student specified possible steps to test the target methods. Additionally, the question clearly states which interfaces require mocking (using Mockito) and provides indications of the available libraries.

While we expected that students would try to generate ready-to-use solutions, we believe this large percentage is concerning. Using ChatGPT-like tools like AI Defenders to quickly and automatically obtain solutions to complex problems might damage students' learning: if a solution is easily available, e.g., from a generative model, students are less likely to spend time trying to find the solution by themselves (we show exactly this point in the next section).

Possible explanations for this behaviour could be that some students do not know how to approach the mocking problem without guidance, so they seek reference solutions to start with via the smart assistant. Instead, other students opted to use the smart assistant to take shortcuts, thus obtaining a solution to their testing problem in the easiest way possible.

**RQ1:** We found that students expect the assistant to be able to provide helpful responses about new topics and useful code examples. Moreover, many students also expect to obtain complete (and correct) tests that kill mutants.

## 5.2 RQ2: To what extent do students find GPT’s responses correct and useful?

After the experiment, we surveyed the students to understand whether AI Defenders met their expectations and whether they used its responses to their advantage. We examined the replies collected in the final survey and cross-checked them with our observations based on analysing students’ questions and GPT’s responses.

We asked students to evaluate how useful the assistant was in performing different tasks, such as answering generic questions or providing sensible code examples, and how often it produced correct responses. We report the results in Figure 7 and Figure 8.

As can be seen in Figure 7, most students (54%) agree that the assistant provided valuable responses to generic questions about testing. In contrast, they are mainly neutral concerning assistant contributions for writing specific tests and mutants. This result confirms that most students were able to exploit the assistant effectively for generic questions (such as the ones related to mocking) but found it difficult to extract useful information to write tests and mutants. Interestingly, students have very different opinions concerning the usefulness of the code examples returned along with the assistant’s responses.

We also asked students to assess the correctness of the responses to their questions. In particular, we requested students to assess the different parts of the responses (i.e., short answer, explanation, and code example when available) separately. As Figure 8 shows, most students (respectively 75% and 70%) find short answers and explanations to be at least often correct, although some students lamented that both sections were too short. We believe these concerns, which are particularly relevant for more theoretical questions, can be addressed by adapting the system prompt and allowing GPT to produce more verbose responses.

Regarding the code examples, 57% of the students believe that they are only sometimes correct, while 25% believe the generated code examples are rarely or never correct. Some students elaborated that the code examples did not work out of the box because they lacked the mocking code entirely or included mocking libraries that Code Defenders does not support.

A qualitative analysis of the questions submitted by the students confirmed the issues reported in the final survey. However, the analysis also revealed that some students coped with this issue by including more details, e.g., about the mocking libraries allowed in Code Defenders, in their questions before re-submitting them. Therefore, we argue that GPT’s inability to generate correct test cases is partially caused by students’ tendency to ask imprecise questions or questions lacking fundamental details. To confirm this hypothesis, we computed the average length of English questions: according to our data, they contain, on average, 16.0 words (with a standard deviation of 16.1). We believe this metric is representative of the simplicity of students’ questions, which limits GPT’s capabilities to provide better responses. We also believe AI Defenders’s design did not influence or impair students’ ability to ask reasonably long questions as its default limit on question size is 1500 words.

As students have mixed opinions about the usefulness of the code examples generated by the smart assistant and many of the generated test cases did not compile, we estimate how many times students could fix broken test cases and make them at least compile.

For this purpose, we counted the number of questions asking for a test of a specific method or mutant that were later followed by a valid test case on that method or mutant written by the same player. We found that in 54.2% of the cases, such questions were not followed by a corresponding valid test. This result suggests that many students were not able to fix GPT’s code or, as confirmed in the final survey, believed fixing the broken tests takes too much time. Our observation is also supported by the failing tests metric reported in Table 2: on average, students with the smart assistant submitted more not-compiling tests (5.14) than students without the assistant (4.22).

**RQ2:** The smart assistant only partially met students’ expectations. It provided useful responses to generic questions about testing but performed poorly while generating test cases, partially due to the brevity of the student’s questions. We also observed that many students could not fix broken tests or avoided doing so and were frustrated and inefficient.

## 5.3 RQ3: Does students’ performance improve using the assistant?

One of the main goals of our study is to understand whether our ChatGPT-like smart assistant could improve students’ performance. We compared students’ average performance metrics (see Section 2) with and without the smart assistant to address this question. We report the summary of the results in Table 2.

From the table, we can see that with respect to all the considered performance metrics, on average, the students who used the smart assistant performed slightly worse than otherwise: they scored fewer points, wrote fewer valid and useful tests and mutants, but wrote more broken tests and mutants. Although the *Wilcoxon signed-rank test* suggests that those results might not be statistically significant ( $p\_value < 0.05$  only for valid tests), we believe that such an outcome is a warning for those who want to employ ChatGPT-like tools for learning to be cautious: without any training, support, and supervision, accessing such systems might cause more harm than good to students.

To shed more light on this critical point, we compared the achieved performance across the two rounds of Code Defenders for each individual student: only 27.0% of the students scored more points using the assistant, only 32.4% wrote more useful tests, and only 46.0% wrote more useful mutants. These data align with the trends observed before and explain the mixed reactions observed in the survey: introducing the assistant had a different impact on various students. Specifically, for 18 students, the smart assistant was a burden and caused them to write fewer useful tests; for 7 students, it was almost noninfluential; finally, for the remaining 12 students it was beneficial.

Our interpretation of these results is that only a small fraction of the students knew how to leverage our ChatGPT-like assistant effectively, whereas the others had unrealistic expectations and failed to ask relevant or detailed enough questions.

**RQ3:** On average, students did not benefit from the smart assistant, as they produced neither better tests nor stronger mutants. However, some students leveraged the power of the GPT model and improved their performance with it.



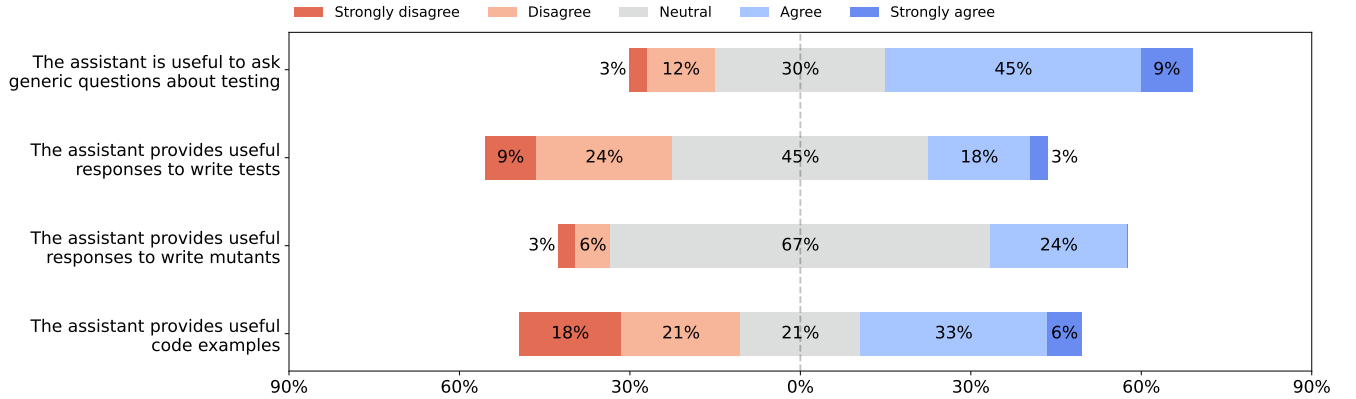


Figure 7: Final survey replies on assistant usefulness

Table 2: Comparison of average performance metrics

	Total Points	Valid Tests	Failing Tests	Useful Tests	Valid Mutants	Failing Mutants	Useful Mutants
With the assistant	8.81	5.54	5.14	2.35	6.16	0.89	4.38
Without the assistant	10.65	7.46	4.22	3.08	7.08	0.84	4.70

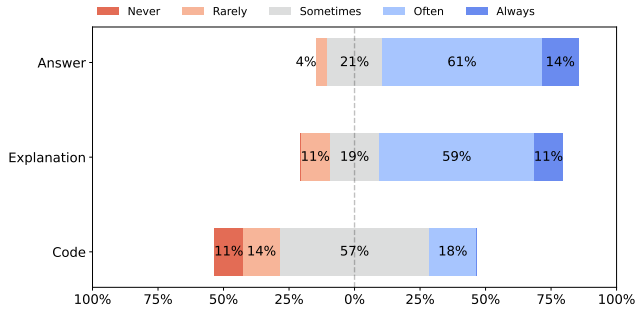


Figure 8: Final survey replies on responses correctness

## 5.4 Discussion

Many observations in our study suggest that students have high expectations from the assistant but cannot exploit it to their advantage. Introducing the smart assistant *as-is* caused many students' performance to degrade and highlighted many anti-patterns in its use that caused unfruitful behaviours. We conclude that giving unrestricted access to such a powerful tool to untrained people, like many university students, might be risky and, to some extent, detrimental to learning.

We argue that the improper use of the tool caused part of the problem, i.e., students could not use it correctly. Therefore, to become more beneficial to them, such smart assistants should mediate the interactions between students and the underlying GPT model. We also conjecture that guiding or training students in using such tools could bring better results since it would allow them to set the right expectations, avoid misuse, and address the other limitations highlighted by our study.

An orthogonal way to increase the benefits of the smart assistant could be to impose some limitations on its usage. For example, the smart assistant could allow only a limited subset of questions or suggest interesting and detailed questions to students. Alternatively, the smart assistant could be designed to actively stimulate students' curiosity, identify struggling students, and provide them with hints and instructions on tackling a certain task (e.g., it could present step-by-step instructions on creating the first test using Mockito). This direction could also be explored by investigating the possibility of developing a custom Large Language Model for Software Testing Education. Being domain-specific, such an LLM could provide more accurate and on-point responses.

Even though restricting access to the GPT model would limit students' freedom, we believe doing so could enhance their learning experience, as some students confirmed in the final survey. We advocate that future work investigates these possibilities to impact the learning experience positively.

## 6 RELATED WORK

Despite being a crucial phase of the software development cycle, software testing is often neglected by programmers, who perceive this activity as repetitive and tedious. Gamification, i.e., the introduction of game elements in non-game contexts, has been proposed to address this issue both in educational and real-world settings [2].

For example, Fu and Clarke extended *WReSTT-CyLE*, a web-based repository of software testing tutorials, with gamification concepts to increase students' engagement [5], and found a positive correlation between students' game points and their final grades in software engineering courses. Straubinger and Fraser introduced gamification aspects directly into the development environment via the *IntelliGame* plugin [16] that monitors the developers' activity in the IDE and automatically challenges developers to write more

tests, achieve increased coverage and produce better test suites. We also integrated AI Defenders in a gamified environment; however, we aimed to support learning by providing guidance and explanations rather than increasing student engagement, motivation, or productivity.

Many recent studies have investigated the applicability of LLMs in the field of software testing. In this area, Schäfer, Li, Jalil and co-authors' work is remarkable. Schäfer et al. proposed *TestPilot* [14] to automatically generate unit tests in JavaScript from a testing skeleton template, the code of the function to be tested and its documentation. As reported, *TestPilot* generated tests with non-trivial assertions 61% of the times and achieved 62% statement coverage. Li et al. proposed *Differential Prompting* [9] to generate failure-inducing test cases with high effectiveness (i.e., 78% of the time). They exploited ChatGPT's ability to infer the behaviour of code samples to create a specification and a correct version of a target buggy program and used them to derive regression oracles. In the context of software testing education, instead, Jalil et al. investigated whether ChatGPT could answer questions and solve programming quizzes on basic software testing concepts taken from a widely used textbook [7]. As their evaluation concluded, ChatGPT correctly answered questions in 49% of the cases and generated satisfying explanations in 40% of the cases.

These studies focused on the LLMs and investigated their performance while solving software testing and education tasks. On the contrary, our study focuses on the students and how they interact with LLMs and aims to understand the potential risks and benefits that this disruptive technology brings to education environments.

## 7 CONCLUSIONS AND FUTURE WORK

Generative AI and Large Language Models, such as ChatGPT, are powerful technologies that are easily accessible to many. They can bring benefits to those students who know how to use them, thus improving their learning process; however, as discussed in this paper, they can be easily misused and quickly become a burden that hinders students' ability to perform well.

In this paper, we report on an empirical study we ran to understand how software testing students interact with a GPT-based smart assistant while learning and practising software testing using Code Defenders. We found that students expect the smart assistant to provide useful responses featuring complete and correct code examples. While students consider the explanations of generic topics they received usually correct and helpful, specific code examples are (too) often incorrect, casting doubts on their effective usefulness for learning. We argue that generating answers that are not useful and low-quality code examples is partially caused by students' inability to ask sufficiently detailed and articulated questions.

To the best of our knowledge, our empirical result is the first investigation using ChatGPT-like tools in the context of learning and practising new testing concepts. While considering the limited size of our study we refrain from drawing strong conclusions, we optimistically believe that tools like our smart assistant can still be beneficial for students' learning if used correctly. Therefore, in the future, we plan to investigate suitable ways to mediate the interaction between the students and the underlying GPT models,

stimulate students' curiosity, and address the limitations of the current implementation highlighted in the study.

## ACKNOWLEDGMENTS

We thank Stephan Lukasczyk and Alexander Degenhart for their support. This work is partially supported by the DFG under grant FR 2955/2-1, "QuestWare: Gamifying the Quest for Software Tests".

## REFERENCES

- [1] Marian Daun and Jennifer Brings. 2023. How ChatGPT Will Change Software Engineering Education. Association for Computing Machinery.
- [2] Gabriela Martins de Jesus, Fabiano Cutigi Ferrari, Daniel de Paula Porto, and Sandra Camargo Pinto Ferraz Fabbri. 2018. Gamification in Software Testing: A Characterization Study. In *Proceedings of the III Brazilian Symposium on Systematic and Automated Software Testing (SAO CARLOS, Brazil) (SAST '18)*. Association for Computing Machinery, 39–48. <https://doi.org/10.1145/3266003.3266007>
- [3] Gordon Fraser, Alessio Gambi, Marvin Kreis, and José Miguel Rojas. 2019. Gamifying a Software Testing Course with Code Defenders. In *Proc. of the ACM Technical Symposium on Computer Science Education (SIGCSE) (SIGCSE'19)*. ACM.
- [4] Steven Freeman, Tim Mackinnon, Nat Pryce, and Joe Walnes. 2004. Mock Roles, Not Objects. 236–246. <https://doi.org/10.1145/1028664.1028765>
- [5] Yujian Fu and Peter J Clarke. 2016. Gamification-based cyber-enabled learning environment of software testing. In *2016 ASEE Annual Conference & Exposition*.
- [6] Jerome Goddard. 2023. Hallucinations in ChatGPT: A Cautionary Tale for Biomedical Researchers. *The American Journal of Medicine* (2023).
- [7] Sajed Jalil, Suzzana Rafi, Thomas D. LaToza, Kevin Moran, and Wing Lam. 2023. ChatGPT and Software Testing Education: Promises & Perils. In *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 4130–4137. <https://doi.org/10.1109/ICSTW58534.2023.00078>
- [8] Benjamin Clegg José Miguel Rojas, Thomas White and Gordon Fraser. 2017. Code Defenders: Crowdsourcing Effective Tests and Subtle Mutants with a Mutation Testing Game. In *Proc. of the International Conference on Software Engineering (ICSE) 2017*. IEEE, 677–688.
- [9] T. Li, W. Zong, Y. Wang, H. Tian, Y. Wang, S. Cheung, and J. Kramer. 2023. Nuances are the Key: Unlocking ChatGPT to Find Failure-Inducing Tests with Differential Prompting. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 14–26. <https://doi.org/10.1109/ASE56229.2023.00089>
- [10] OpenAI. 2022. Introducing ChatGPT. <https://openai.com/blog/chatgpt>.
- [11] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Chapter Six - Mutation Testing Advances: An Analysis and Survey. *Advances in Computers*, Vol. 112. Elsevier, 275–378. <https://doi.org/10.1016/bbs.adcom.2018.03.015>
- [12] Md. Mostafizer Rahman and Yutaka Watanobe. 2023. ChatGPT for Education and Research: Opportunities, Threats, and Strategies. *Applied Sciences* 13, 9 (2023). <https://doi.org/10.3390/app13095783>
- [13] José Miguel Rojas and Gordon Fraser. 2016. Code Defenders: A Mutation Testing Game. In *Proc. of The 11th International Workshop on Mutation Analysis*. IEEE, 162–167.
- [14] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Transactions on Software Engineering* 50, 1 (2024), 85–105. <https://doi.org/10.1109/TSE.2023.3334955>
- [15] Sandro Speth, Niklas Meißner, and Steffen Becker. 2023. Investigating the Use of AI-Generated Exercises for Beginner and Intermediate Programming Courses: A ChatGPT Case Study. In *2023 IEEE 35th International Conference on Software Engineering Education and Training (CSE&T)*. 142–146. <https://doi.org/10.1109/CSEET58097.2023.00030>
- [16] Philipp Straubinger and Gordon Fraser. 2024. Improving Testing Behavior by Gamifying Intellij. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (Lisbon, Portugal) (ICSE '24)*. Association for Computing Machinery, Article 49, 13 pages. <https://doi.org/10.1145/3597503.3623339>
- [17] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems (New Orleans, LA, USA) (CHI EA '22)*. Association for Computing Machinery, Article 332, 7 pages. <https://doi.org/10.1145/3491101.3519665>
- [18] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elashar, Jesse Spencer-Smith, and Douglas C. Schmidt. 2023. A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT. arXiv:2302.11382