The final publication is available at Springer via https://doi.org/10.1007/978-3-031-80889-0_17

Software System Testing Assisted by Large Language Models: An Exploratory Study

Cristian Augusto [0000-0001-6140-1375] [1], Jesús Morán [0000-0002-7544-3901] [1], Antonia Bertolino [0000-0001-8749-1356] [2], Claudio de la Riva [0000-0001-5592-9683] [1], Javier Tuya [0000-0002-1091-934X] [1]

[1] Computer Science Department, University of Oviedo, Gijón,
[2] ISTI-CNR, Consiglio Nazionale delle Ricerche, Pisa, Italy

[1]{augustocristian, moranjesus, claudio, tuya}@uniovi.es
[2] antonia.bertolino@isti.cnr.it

Abstract. Large language models (*LLMs*) based on transformer architecture have revolutionized natural language processing (*NLP*), demonstrating excellent capabilities in understanding and generating human-like text. In Software Engineering, *LLMs* have been applied in code generation, documentation, and report writing tasks, to support the developer and reduce the amount of manual work. In Software Testing, one of the cornerstones of Software Engineering, *LLMs* have been explored for generating test code, test inputs, automating the oracle process or generating test scenarios. However, their application to high-level testing stages such as system testing, in which a deep knowledge of the business and the technological stack is needed, remains largely unexplored. This paper presents an exploratory study about how *LLMs* can support system test development. Given that *LLM* performance depends on input data quality, the study focuses on how to query general purpose *LLMs* to first obtain test scenarios and then derive test cases from them. The study evaluates two popular *LLMs* (*GPT-4o and GPT-4o-mini*), using as a benchmark a European project demonstrator. The study compares two different prompt strategies and employs well-established prompt patterns, showing promising results as well as room for improvement in the application of *LLMs* to support system testing.

Keywords: Large Language Model, Software Testing, System Testing, Test Cases, Test Scenarios

1 Introduction

Large Language Models (onwards referred to as *LLMs*) based on transformer architecture have emerged as one of the biggest technological disruptions of recent years in the field of natural language processing (*NLP*). In a nutshell, *LLMs* are deep neural networks trained on a huge amount of data, from which they acquire an astonishing capability to understand and generate human-like text. State-of-the-art *LLMs* show such

"*intelligent and rational*" capabilities in imitating human performances that in a recent experiment they have been able to pass the Turing test with a 50% success rate [1].

LLMs are rapidly transforming the technological landscape while also attracting increasing attention from the media and raising society expectations on their potential benefits. *LLMs* are already widely employed in industry, especially in healthcare, education and financial services [2]. Typical *LLM* applications include for example the creation of conversational agents (bots), empowered support to user's performing tasks or process automation [3], moving towards the so-called industry 6.0 age [4].

In terms of market expansion, even though the estimated (compound annual) growth rates differ largely (from the 33.2% by MarketsandMarkets [5] up to the 79.8% by Pragma Market Research [6]) the analysts concur in predicting steady growing worldwide impact until 2030.

LLMs have also crashed into *Software Engineering (SE)* [7], hinting at several potential breakthroughs in addressing *SE* challenges that can be reformulated in terms of data, code or text analysis [8]. To date, *LLMs* have achieved quite promising results in assisting the developer to generate code, documentation, and reports [9], in improving the explainability of the code itself or its patches [10], and in other tasks considered repetitive and less valuable (so-called "*toils*" [11]).

In Software Testing, one of the cornerstones of the *SE* field, several works have explored how to use *LLMs* to generate test code [12] or also to generate test scenarios [13]. However, as we discuss in the Related Work section, the focus has been mostly on the Unit Test level: as noticed by Ozkaya in a recent editorial about the application of *LLMs* to *SE* tasks, "*Generating unit tests is one of the tasks where developers shortcut the most*." [7].

In this work, we aim to investigate the capability of *LLMs* to support software testing at system level. System test cases validate the interaction among the different system components or the user interaction with the application. The system test suite development process is usually expensive because it requires a deep knowledge of the business context of the application and knowledge about the technological stack on which the application relies. Therefore, if *LLMs* could be leveraged to help the generation of effective system test cases, this could bring substantial benefits to the whole testing process. To the best of our knowledge, the application of *LLMs* to support system testing remains largely unexplored: some works have scraped the surface by evaluating *LLM* support to test specific types of applications, or to generate test inputs, e.g., for mobile applications. However, no previous work has used *LLMs* to automate the whole path of deriving system test cases from user requirements.

The approach we experiment here considers two different artifacts related to system testing, namely test scenarios and test cases. The former are taken as an input for the latter, in accordance with the *ISO 29119* standard, in which test scenarios are defined as *"the situations or settings for a test item used as a basis for generating the test cases"* [14]. Thus, in this work we first evaluate the support of *LLMs* for deriving test scenarios, which provide a high-level stepwise description of the system tests; afterwards, we also use the *LLMs* to generate the test cases, i.e., the test code that corresponds to a given test scenario.

While remarking on the excellence of *LLMs* for a wide range of tasks, several authors warn that the *LLM* performance is directly related to the quality of their input [15]. In *LLMs* we differentiate between two types of input data: (1) the training data used during the creation of the model and (2) the query data used to prompt the model. Currently, most approaches employ general-purpose pre-trained *LLMs*, tailored for specific tasks through appropriate query data to *prompt* the model [16]. Therefore, to investigate how *LLMs* perform in assisting the system test process, we need to explore how these general purpose pre-trained *LLMs* should be prompted. To achieve this objective, this article uses both the data of a real-world application and its test suite to evaluate how two of the most popular state-of-the-art pre-trained models (*GPT-4o* and *GPT-4o-mini)*, perform when asked to assist the system testing process in generating test scenarios and coding the system test cases.

Although we cannot generalize our conclusions outside the employed evaluation subject, our study provides promising results for both tasks, hinting at the opportunity of leveraging *LLMs* to assist the entire system testing process, from test scenario derivation until test cases development.

In our exploratory study we have observed that the LLMs can support the tester during the E2E system process, facilitating their task. According to our initial evaluation, the LLM test scenario generation covered the user requirements extensively (up to 100%). However, the generation of E2E test code requires to slightly change a few lines of the code (up to 29%).

The rest of the paper is structured as follows: *Section 2* provides the related work. The exploratory study design is presented in *Section 3*. *Section 4* presents the evaluation and results while *Section 5* presents the threats to validity. Finally, *Section 6* presents the conclusions and future work.

2 Related Work

In this section we review the literature related to our paper. The related work belongs primarily to two fields: (1) Large Language Models and Prompt Engineering and (2) Large Language Models applied to Software Engineering and Software Testing.

2.1 Large Language Models and Prompt Engineering

Large Language Models (*LLMs*) based on neural networks were introduced in the 80s [17]–[19] for Natural Language Processing (NLP). Later, the introduction of the transformer architecture [20], triggered an explosion in the number of multi-purpose models with reasoning-like capabilities like *PALM* [21], *Llama* [22], or *GPT-4* [23]. These models follow the "*Pre-train, Prompt and Predict*" [16] paradigm, in which a general purpose *LLM* is adapted to a new concrete task through an adequate *prompt*. Therefore, the process of designing and refining the prompts for a pre-trained model, also known as *prompt engineering*, has attracted the interest of both academia and industry; several authors have proposed patterns to accomplish different related tasks [24], [25], various *prompt* strategies [26]–[28], as well as development tools and repositories [29].

Our exploratory study draws upon all these works, employing state-of-the-art transformer architecture pretrained models [23], using the *pre-train, prompt and predict* paradigm [16]. As described in *Section 3.2*, we employ state-of-the-art patterns [24] to design and create our *prompts* following the best practices [25], and use the *Few-Shot* [28] and *Chain of Thought* [27] prompt techniques.

2.2 Large Language Models applied to Software Engineering and Software Testing.

In recent years, the increasing popularity of *LLMs* has exploded in a wide range of applications in Software Engineering including code generation, code explainability, and more in general for reducing the manual effort in repetitive tasks or improving/easing difficult processes such as Software Testing. *LLM-assisted* generation of code has been popularized through the inclusion of programming assistants like *Microsoft Copilot* [30]. This type of assistants has attracted the interest of academia, who focused on the impact of its *hallucinations* and the quality and usefulness of its code [30]–[32]. Code explainability using *LLMs* has been addressed through the explainability of the code itself [33], or the explainability of failures, debugging [34], generating documentation [35], and reports [9], [10]. In Software Testing, part of the effort has been put into unit testing for generating test cases, migrating testing code or generating test scenarios [12], [13], [36], [37]. In other testing levels, like System Testing, the literature has focused on generating test inputs/data for mobile/UI testing, such as the necessary human interaction or text inputs [38] or test inputs for other types of software (e.g., simulators) [39]. In mobile testing LLMs have been applied to generate user UI interactions (testing scripts) with the GUI information, generate the [15] navigation through the application using natural language test cases [40] or [37] generate mobile test code from natural language. In GUI testing, some authors [34] have also applied *LLMs* to migrate test scripts between different platforms and apps.

Finally, other works that explore how to generate reports or models [9], [10], [35] provide us with insights on how to address prompt engineering as well as its evaluation in Software Engineering.

Some works share similarities with our approach: [40] has the same objective but the test cases provided as input, and the outputs are direct descriptions of UI interactions (e.g., *press button "A"*). In [15] the authors propose using the GUI as input, but the output is again a natural language description, e.g., *"Operation: Scroll, Widget: Menu"*. In contrast, our approach uses as input the user requirements, generates scenarios, and from these scenarios with test case examples generates the testing scripts that with slight adjustments can be executed directly against the application.

The closest works to this paper are I) the generation of test scenarios using system requirements proposed by [13] and II) the test script generation based on natural language specifications proposed by [37]. Our work attempts to go a step further by exploring the whole process: we first generate the test scenarios from the user requirements, and then generate the test code from them. In perspective, we aim at a fully supported system testing approach that can benefit from *LLM* assistance, and in this work we take the first steps in this direction.

3 Exploratory Study Design

3.1 Overview

The process followed in our exploratory study, depicted in **Fig. 1**, consists of two subprocesses: the generation of test scenarios (left side of **Fig. 1**) and the generation of test cases (right side of **Fig. 1**). These two subprocesses are described below:

- The generation of test scenarios starts by giving as input the user requirements of the application to generate test scenarios [14]. More precisely, the user requirements (**Fig. 1**, ①), as well as serving an example of a test scenario, (**Fig. 1**, ②) are provided to generate the prompts of two different prompt techniques: *Few-Shot-prompting* [41] (**Fig. 1**, ③) and *Few-Shot + Chain of Thought prompting* (**Fig. 1**, ④). We generate two different prompts that are given to two *LLMs*: *GPT-4o* and *GPT-4o-mini* to generate the different scenarios (four sets in total) for each *prompt* technique (one different prompt for each technique) and *LLM* model (**Fig. 1**, ⑤-⑧).
- The generation of test cases uses as input the test scenarios generated in the previous subprocess (**Fig. 1**, ⑨) and some examples of system test cases (**Fig. 1**,⑩). We generate two different prompts, again one for each technique: *Few-Shot* and *Few Shot + Chain of Thought*. The *prompts* are provided to the *GPT-4o* and *GPT-4o-mini* models that eventually generate four different sets of test cases (**Fig. 1**,⑬-⑯).

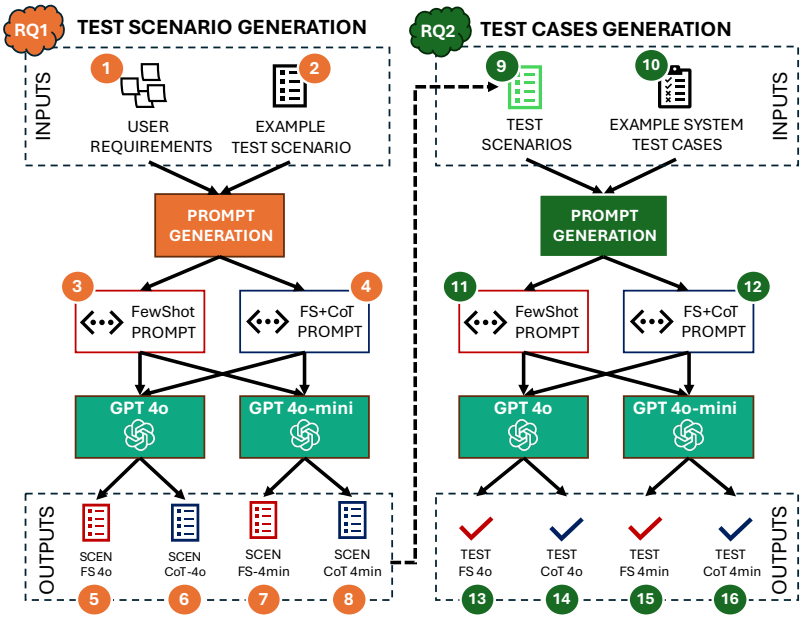

Fig. 1 Overview of exploration study

In accordance with the two subprocesses above described, our exploratory study aims to answer two Research Questions:

RQ1: How do *LLMs* perform in generating system test scenarios from user requirements?

RQ2: How do *LLMs* perform in generating test cases from test scenarios?

In the following subsections, we describe the set-up and the design of our exploratory study.

3.2 Study subject and evaluation set-up

As an **evaluation subject,** we use the real-world application *FullTeaching* [42], a demonstrator of the Horizon 2020 European project ElasTest [43]. *FullTeaching* is a web application that provides an online teaching platform to impart classes, publish materials, and enable the interaction between teachers and students. Specifically, we employ from this web application:

- The user requirements (translated from Spanish) [44] used during the development of the *FullTeaching* web application and available in the *FullTeaching* project documentation [44]. Precisely, they consist of 39 natural language requirements that cover functionalities such as videocall or course creation.
- The system test suite of this application available in the GIIS repository [45], composed of 21 Java Selenium test cases that cover different test scenarios.

Below is an example of the user requirements, the full list is available in the **replication package** [46]: "(*16) The teacher can add students to a course in different ways: (16.1) using their unique email address, (16.2) using multiple email addresses, or (16.3) using a file (txt, excel, Word, PDF…).*"

In **Table 1** below, we provide the user requirements covered (shown in the rows using their original labelling from 1.0 to 16.0) and the different system test cases (shown in the different rows). This user requirement coverage was derived by reaching consensus among the authors. We see that the 21 test cases available from *FullTeaching* cover almost 80% of the 39 user requirements, leaving only uncovered (8) the registration, (7.3) the muting of the teacher audio by the students, (7.7) the cancellation of a requested voice turn, (11) the edition of a class attributes at any time, (12) the profile modification and (14) the captcha test during the registration. This test suite was developed and then extended-migrated to the ElasTest repository. We do not know why the author did not test the whole functionality, we can only guess that this was perhaps due to time constraints.

As a ***Large Language Model***, we have studied the current market and decided to select two of the latest models offered by the leader industry *OpenAI* [47], the models concisely are *GPT-4o* (v2024-05-13) and *GPT-4o-mini* (v2024-07-18). We execute the prompts using the *OpenAI* API, setting the *temperature* of the models to 0.2 [48] to improve the repeatability of the results. This parameter controls the model's randomness, a value of 0 selects the highest probability words, while higher temperatures increase creativity and the chance of '*hallucinations*'.

As **evaluation metrics,** concerning RQ1 we consider the test coverage metric: "*number of requirements covered by executed tests*" [14]. We adapt this metric to the

case of the *LLM-generated* test scenarios, i.e., we evaluate which and how many user requirements are covered by the *LLM-generated* test scenarios instead of by the executed test cases. This measure of coverage (in percentage) is compared against the requirement coverage achieved by the baseline, i.e., which and how many user requirements are covered by the *FullTeaching* test suite (shown in **Table 1**). Requirements 1, 2, 3, and 9 covered by most of the test cases correspond to: the user can see the courses enrolled in, the user can access a course, teachers can create a course, and the user can log in to the system, respectively. Concerning RQ2, considering the test cases generated by *LLM*, we measure the effort required to make the generated test code pass (unless we ascertain that they fail because of a bug in the application under test). As a proxy measure of this effort, we use the average number of code changes needed until the test code successfully executes, counting the number of lines of code modified.

In the replication package we provide a series of Java scripts using the *OpenAI API* to query and obtain the answers of the models, the *prompt* templates, strategies, and the findings of our exploratory study. The replication package is available in our *GitHub* repository [46].

Table 1 User Requirements FullTeaching traceability matrix

E2E Test Case	**Requirement Covered**
oneToOneChatInSessionChrome	1, 2, 6.1, 6.4, 6.5, 6.8, 7.1, 7.4, 7.5,9
courseRestOperations	1, 2, 3, 9, 10
courseInfoRestOperations	1, 2, 3, 9, 10, 13
sessionRestOperations	1, 2, 3, 4.1, 4.2, 4.3, 9
forumRestOperations	1, 2, 3, 4.6, 5.1, 5.2, 9
filesRestOperations	1, 2, 3, 5.5, 5.3, 9
attendersRestOperations	1, 2, 3, 4.4, 9, 16.1
sessionTest	1, 2, 6.1, 6.4, 7.1, 7.4, 9
oneToOneVideoAudioSessionChrome	1, 2, 4.1, 4.2, 4.3, 6.1, 6.2, 6.3, 6.4, 6.6, 6.7, 6.8, 7.1, 7.2, 7.4, 7.6, 7.8, 9
studentCourseMainTest	1, 2, 9, 15
teacherCourseMainTest	1, 2, 9, 15
teacherCreateAndDeleteCourseTest	1, 2, 3, 5.1,9,10, 13
teacherEditCourseValues	1, 2, 3, 9
teacherDeleteCourseTest	1, 2, 9
forumLoadEntriesTest	1, 2, 5.2, 9
forumNewEntryTest	1, 2, 5.2, 9
forumNewCommentTest	1, 2, 5.2, 9
forumNewReply2CommentTest	1, 2, 9
spiderLoggedTest	1, 2, 9
spiderUnLoggedTest	9
loginTest	9

3.3 *Prompt* creation and refinement

To create the *prompts* to answer the two research questions we first establish a base *prompt* for *Few-Shot*, and then extend it with the necessary statements to also apply the *Chain-of-Thought* prompt technique. *Few-Shot prompting* is a technique that enables in-context learning by providing the *LLM* with examples in the *prompt* (e.g., different assertions in one language if we are asking for a concrete assertion). *Chain of Thought* prompting aims to enable the reasoning capabilities of the model by explicitly requiring the intermediate steps. Precisely, the *prompt* creation was carried out as follows:

***Few-shot Prompts*:**
To generate the test scenarios using *Few-Shot* we create our *prompt* using the *Recipe prompt* pattern [24], as we find that a test scenario has similarities to a recipe: several ordered tasks, an expected output and also incorrect states. The *prompt* structure, as well as the contextual statements to address the test scenario generation, is as follows:

I) *"I would like to generate test scenarios for system testing*
I know that I need to fulfill the user requirements: ''' Ⓐ {{UserRequirements}} '''
Provide a complete sequence of steps for each scenario and the expected outputs.
Fill in any missing steps
Identify any unnecessary steps.
Examples of a test scenario: ''' Ⓑ {{ExampleTestScenario}} ''''

The *prompt* has available two placeholders: (1) *UserRequirements* (**I**, Ⓐ) contains the application user requirements, while (2) *ExampleTestScenario* (**I**, Ⓑ) contains from one to several examples of test scenarios.

To generate the system test cases, with the *Few-Shot prompt* technique we use the Context Manager *prompt* pattern [24]. This pattern allows us to delimitate the context of the *LLM*, focusing on the system test cases provided as examples and the test scenarios. The *prompt* is the following:

II) *"When generating system test that covers Ⓐ {{Functionality}}*
Please consider the following test scenarios: ''' Ⓑ {{TestScenarios}} ''''
and the following system test examples: ''' © {{SystemTestExamples}} ''
Don't generate the whole test suite, only the required test case."

This *prompt* presents three different placeholders: (1) *Functionality* (**II**, Ⓐ) expects the test functionality to be covered by the generated test case. In the prompt, the *Functionality* placeholder is substituted with a brief, high-level statement that describes what is going to be tested such as: "*User enrolls into a course*" or "*Teacher login and create a course*". In most of the cases, we use the title of the test scenario to be covered, or a summary of it. The (2) *TestScenarios* (**II**, Ⓑ) placeholder expects all the test scenarios, and finally (3) *SystemTestExamples* (**II**, ©) expects the system test cases given as examples to the *LLM*. This *prompt* has been refined based on our experience generating system test code: "*Don't generate the whole test suite, only the required test case*"

delimitates the generation of code, avoiding that in some executions the LLM generates more cases than required.

Few-shot + Chain of Thought Prompts

To create the *prompts* for *Chain of Thought*, we extend the *Few-Shot prompts* by adding the phrase *"Let's think step by step"*, which has been proved as a robust *"reasoning enabler"* for instructive outputs [28].

To generate the test scenarios, we include as the first statement the following line:

> **III)** *"Let's think step by step, describe the solution and remark which user requirements are covered"*

To ensure that the reasoning capabilities of the *LLM* are focused on the coverage of the different user requirements, we ask to highlight which user requirements are covered by each test scenario.

For the system test generation, we include the following statements to the *Few-Shot prompt*:

> **IV)** *"Let's think step by step, describe the solution by breaking it down into a task list for then generate the code"*

We add this suggestion to establish the steps of the test case prior to the code generation, as, according to several *prompt* engineering demonstrations and tutorials [46], this improves the expected output.

4 Exploratory Study Execution and Results

The following subsections detail how the exploratory study has been carried out. Section 4.1 and Section 4.2 present the evaluation, respectively, for RQ1 and RQ2.

4.1 [RQ1]: Generating test scenarios from user requirements

To answer the first research question, we employ the prompts presented in the previous section, and fulfill the *prompt UserRequirements* (**I**, Ⓐ) placeholder with the entire *FullTeaching* user requirements and the *ExampleTestScenario* (**I**, Ⓑ) with a test scenario of another business context (bank transaction management platform).

We evaluate the results achieved by the *LLMs* using *Few-Shot* and *Chain-of-Thought* for both models (*GPT-4o and GPT-4o-mini*). Thus, we refer to the right side of **Fig. 1** in which we obtain four different sets of test scenarios denoted as ⑤-⑧. Moreover, as the results provided by the *LLM* are not deterministic, we execute each thread up to 5 times, thereby obtaining 20 sets of test scenarios in total.

As anticipated, we measure for each set of test scenarios the coverage of user requirements (in percentage), and then we also evaluate the reliability of the test scenario generation approach by comparing for each of the four threads the results of the five executions. Our initial findings between the different executions are that for all the

prompting techniques and models studied, the test scenarios are quite consistent (even more than we expected, see **Table 2**).

Table 2 Requirements coverage the different executions, models, and prompt techniques

Prompt Technique	Model	Execution					AVG	MDN	SD
		1	2	3	4	5			
Few-Shot+ CoT	GPT-4o	100	100	100	100	100	100	100	0
Few -Shot	GPT-4o	100	100	100	94.9	100	99.0	100	2.3
Few-Shot+ CoT	GPT-4o-mini	87.2	97.4	87.2	94.9	92.3	91.8	92.3	4.6
Few-Shot	GPT-4o-mini	94.9	97.4	89.7	97.4	92.3	94.4	94.9	3.3

Small differences among the five sets of test scenarios could include, for instance: one scenario may correspond to the merging between two scenarios of another execution, or it can have different text styles, or be named in a different way: "*User enrollment courses*" vs "*Viewing enrolled courses*".

In terms of user requirements coverage, the generated test scenarios have a low standard deviation: 2.3 *GPT-4o/Few-Shot*, 0.0 *GPT-4o/CoT*, 3.3 *GPT-4o-mini*/*Few-Shot* and 4.6 *GPT-4o-mini/CoT*.

Using the *GPT-4o* model with the *Few-Shot prompt* technique, the average in user coverage is 99.0% and the median 100%. With the *Few-Shot + CoT prompt* technique, the average in user coverage is higher (100%) and the median is the same.

Using the *GPT-4mini* model with the *Few-Shot prompt* technique, the average in user requirements coverage is 94.4% and the median 94.9%. With the *Few-Shot + CoT prompt* technique, decreases both the average of coverage (91.8%) and the median (92.3%).

Based on the observed results, we can answer RQ1 as:

> Overall, our findings indicate that model *GPT-4o* slightly outperforms model *GPT-4o-mini* in generating test scenarios. Furthermore, the use of *Few-Shot* with the *Chain of Thought* prompting improves the user requirements coverage in the *GPT-4o-mini* model. Both models achieve better user requirements coverage in all their executions than the baseline 79.46%, generating scenarios that cover functionalities not explored by the *FullTeaching* test suite.

4.2 [RQ2]: Generating system test cases from test scenarios.

To answer RQ2, we employ the above-presented *prompts (***II** *and* **IV***)*, in which we have to fulfill the placeholders: *Functionality*, *TestScenarios* and *SystemTestExamples*.

- With respect to *TestScenarios* (**II**, Ⓑ), given that five sets were generated for each configuration in the earlier phase of our study, it is now crucial to select one single set from the 20 available test scenarios. To this end, we considered the previously defined test coverage metric over the user requirements, and we opted for the set of

test scenarios that achieves the highest coverage; in case of equal coverage value, we prioritize the set with fewer scenarios. Based on these criteria, the selected set of Test Scenarios was one among the 5 sets produced by the *GPT-4o* model using *Few-Shot + CoT prompt* technique.

- Concerning *SystemTestExamples* (**II**, ©), in our study we can use some test cases of those available in the *FullTeaching* test suite (which of course have not been derived from the *LLM* test scenarios). To select which test cases to provide in the prompt, once again we refer to the coverage of the user requirements. Since we have traced the coverage of user requirements by both the test scenarios and the test cases, we can evaluate the relation between a test case and a test scenario comparing their respective coverages. Intuitively, if a test case yields an identical, or very similar, coverage spectrum as a test scenario, we can consider that the test case implements that test scenario. Thus, for each possible pair of a test scenario and a *FullTeaching* test case, we computed their *Levenshtein* distance comparing their respective coverage of the user requirements. We did not consider trivial test cases and scenarios (e.g., login, covered by most of the suite test case). These test cases and scenarios are not considered because the exact code-test methods are provided with all test cases provided by example in the *prompt*, being less challenging for the model. For example, the login test is a simple invocation to *slowLogin (user,password)*, present in all test cases already in the *prompt*. Finally, we selected 4 different test cases as the ones yielding the lowest distance measures from the test scenarios; these 4 test cases, and the 4 test scenarios to which they are close, are referred to as the *Levenshtein* set. We limited the selection to four test cases because we aimed to employ a cross-validation technique, which would have become very costly and time-demanding with a higher number of test cases. The cross-validation technique is described below.
- Concerning *Functionality (***II**, Ⓐ*)*, we refer to the titles of the test scenarios in the *Levenshtein* set, and we select one of them according to the cross-validation approach as described below.

In every execution of the LLMs, all the test scenarios in the selected set were used to fulfill the *TestScenarios (***II** , Ⓑ) placeholder. Instead, the *Functionality (***II**, Ⓐ*)*, and *SystemTestExamples (***II** , ©) take different values according to the cross-validation approach. The cross-validation was performed as follows for each *prompt* technique and model: we remove one test case from the *Levenshtein* set and provide as input the three remaining test cases, asking the LLMs to generate a test case that covers the functionality corresponding to the title of the test scenario closely covered by the one that has been removed.

As anticipated, to assess *LLM* results for the different models and *prompt* techniques, we refer to the effort required to make the test code pass, in terms of the number of lines of code modified. We execute the generated test cases over the *FullTeaching* application, and we manually made the minimal changes to make the test case work/pass. **Table 3** shows the number of changes in the code performed, the average, and the percentage in average of lines changed for the different test cases (A-C) with the two models (*4o* and 4o-mini) and two prompting techniques (Few-shot and Few-Shot+CoT):

Table 3 Total and average code changes in the different test cases

Model	Prompt Technique	(A) View Courses	(B) View Classes	(C) Create Course	(D) View Calendar	AVG	% AVG
GPT-4o	Few Shot (FS)	2	5	14	3	6	22.64 %
GPT-4o	Few Shot + CoT	6	9	14	4	8.25	31.13 %
GPT-4omini	Few Shot	5	7	18 (14H)	6 (1H)	9	31.30 %
GPT-4omini	Few Shot + CoT	8 (6H)	10	15 (8H)	4	9.25	34.9 %

The results show that using the *GPT-4o* model with the *Few-Shot prompt* technique, the generated test case requires a median of 4 modifications and 6 modifications on average between the different tests generated in the cross validation. Using the *Few-Shot + CoT prompt* technique, the test case requires 7.5 modification in median and 8.25 modifications on average.

Using the *GPT-4mini* model with the *Few-Shot prompt* technique, the generated test case requires 6 modifications in median and 9 modifications on average. Using the *Few-Shot + CoT prompt* technique, the test case requires 7 modifications in median and 9.25 modifications on average.

We have mostly observed two different modification types: in several executions the LLM indicates that it is not sure about a certain value-method and asks the tester for its completion; in other cases, the LLM uses a non-existent method, class, or id (**Table 3** (H-*Hallucination*). The slight modifications required are easy to perform, for instance adjusting a UI identifier to its correct value, replacing methods that do not exist with the correct ones, or adding an annotation that was missed by the LLM. Mostly, we observe that up to 77.5% on average of the generated lines code can be directly used.

The *GPT-4o* model is more prone to use those incorrect object identifiers or methods that do not exist (*Hallucination*). On the other hand, the *GPT-4o-mini* requires more modifications, but some of them are required by the model itself (e.g., "*adjust this ID with the submit button id"*, *"Set password and user to the correct values"*)
Based on the observed results, we can answer RQ2 as:

> In general, our findings indicate that model *GPT-4o* outperforms the *GPT-4o-mini* in generating system test cases from the test scenarios. With reference to the *prompt* technique, *Few-Shot* outperforms the inclusion of *Chain of Thought*, requiring less modifications. In general, the code that needs to be adjusted is around 29% of the total generated code with a standard deviation of 5%.

5 Threats to validity

Notwithstanding our diligent endeavors, the validity of the findings for the exploratory study described above remains susceptible to various threats. We acknowledge the existence of the following types [49]:

Internal validity: threats to internal validity lie in possible biases of our exploratory study such that the properties measured over the observed outcomes are not produced by the *LLMs* but are due to other confounding factors. To mitigate potential internal threats, we employed a test suite and the user requirements of a real system used as a demonstrator in a European project as subject of evaluation. Another possible source of subjectivity is the manual calculation of the test scenarios coverage matrix in the baseline and the RQ1; to mitigate this, two authors originally calculated the traceability of the scenarios over the user requirements and then the coverage table was revised and discussed by all authors until a general consensus was reached.

Construct validity: the threats to construct validity are concerned with the validity of the settings of our study procedure. The main external threat to validity is the non-determinism of the *LLM* itself, by which the output of the selected model can differ between executions. In the test generation, we tried to mitigate it by repeating the experiments five times and comparing the outputs across the five test scenarios to minimize the impact of the randomness. In the test generation exploratory study, we performed a cross validation and calculated the average of the number of modifications to deal with this not determinism. Another construct threat relies on the data that the *OpenAI* models use for training. It is impossible to know if those models were already pretrained with the *FullTeaching* code or test suite, meaning that the solution could be overfitted or biased.

External validity: this type of threats refers to the generalizability of the observations. As we only conducted one exploratory study on one subject, we cannot of course make conclusions about the validity of *LLM*-assisted system testing for other differing contexts and applications. While the results are promising, we warn that more experiments are needed to draw more general conclusions. As is well known, the performance of *LLMs* strictly depends on the *prompts* given to them. Thus, even for the *FullTeaching* application, we cannot exclude that different *prompts* provided by different testers could obtain quite different performances. We tried to mitigate this threat by selecting state-of-the-art *prompting* patterns and strategies for the *prompting* creation process, reducing the model temperature to minimize hallucinations, and fix the OpenAI model versions.

Reliability: to tackle this issue and ensure reproducibility by fellow researchers, we provide access to the user execution data, various configurations applied, and the data used as input into a replication package [46]. The replication package also includes the different formatted LLMs outputs and the changes performed into them (RQ2) to make the test cases pass-work.

6 Conclusions and future work

LLMs arise as a promising support tool to complement the system test process. This exploratory study has remarked that during the test scenario generation, LLMs can provide an initial set of test scenarios that cover most of the user requirements. The generated test scenarios show room for improvement, for instance by reducing the number

of test scenarios by merging several into one (e.g., one test scenario can check the enrolled courses as well as the classes of this course).

Deriving system test cases from the test scenarios poses more challenges. In general, we have observed that the *LLMs* output provides correct test skeleton following the test steps of the scenarios but tend to invent (*hallucinate*) with the identifiers of the web elements or the methods created to support the test cases (e.g., navigate to main menu method). In some cases (mostly the least powerful model) the output explicitly indicates that the tester should tune these parameters, but in other cases the LLM generates method calls or identifiers that do not exist.

Overall, our findings, in line with the community opinion, show that *LLMs* are a great tool to reduce the amount of manual work, but must be supervised by a human tester to reduce the impact of hallucinations.

This is a preliminary work, and several lines of future work have been opened due to the promising results. The most prominent research line entails comparing our approach with the state-of-the-art tools not employing generative AI, as well as evaluating our approach with more models (different to the GPT-based family) with different tunings (e.g., different temperature values or introducing embeddings). We also plan to extend and evaluate the performance of *LLMs* in assisting system testing by improving the preliminary approach employed in this first study. We intend to explore how the *prompting* techniques could be improved to achieve better results in terms of test effectiveness, which has not been covered here. We also need to introduce some approach to assess the efficiency of the LLM-assisted process. Finally, we also intend to explore if and how *LLMs* could support the generation of negative (robustness) test cases.

Acknowledgments:
We would like to thank Alessio Ferrari for his help and guidance throughout our first work on this topic. We also want to extend our gratitude to the URJC ElasTest/FullTeaching team for their continuous support, especially Oscar, Pablo, and Patxi. This work was supported in part by the project PID2022-137646OB-C32 under Grant MCIN/ AEI/10.13039/501100011033/FEDER, UE, and in part by the project MASE RDS-PTR_22_24_P2.1 Cybersecurity (Italy).

References

[1] C. Jones and B. Bergen, “Does GPT-4 Pass the Turing Test?,” 2023, [Online]. Available: http://arxiv.org/abs/2310.20216

[2] S. Raman, “The Rise of AI-Powered Applications: Large Language Models in Modern Business,” 2023. https://www.computer.org/publications/tech-news/trends/large-language-models-in-modern-business (accessed Aug. 01, 2024).

[3] S. Minaee *et al.*, “Large Language Models: A Survey,” 2024, [Online]. Available: http://arxiv.org/abs/2402.06196

[4] A. S. Duggal *et al.*, “A sequential roadmap to Industry 6.0: Exploring future manufacturing trends,” *IET Commun.*, vol. 16, no. 5, pp. 521–531, Mar. 2022, doi: 10.1049/CMU2.12284.

[5] MarketsandMarkets, “Large Language Model Market Size And Share Report, 2030,” 2024. Accessed: Aug. 01, 2024. [Online]. Available: https://www.grandviewresearch.com/industry-analysis/large-language-model-llm-market-report

[6] P. M. Research, “Global Large Language Model (LLM) Market Size, Share, Growth Drivers, Competitive Analysis, Recent Trends & Developments, and Demand Forecast To 2030,” 2024. Accessed: Aug. 01, 2024. [Online]. Available: https://www.pragmamarketresearch.com/reports/121032/large-language-model-llm-market-size

[7] I. Ozkaya, “Application of Large Language Models to Software Engineering Tasks: Opportunities, Risks, and Implications,” *IEEE Softw.*, vol. 40, no. 3, pp. 4–8, 2023, doi: 10.1109/MS.2023.3248401.

[8] X. Hou *et al.*, “Large Language Models for Software Engineering: A Systematic Literature Review,” vol. X, no. December, pp. 1–79, 2023, [Online]. Available: http://arxiv.org/abs/2308.10620

[9] P. Jin *et al.*, “Assess and Summarize: Improve Outage Understanding with Large Language Models,” *ESEC/FSE 2023 - Proc. 31st ACM Jt. Meet. Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, pp. 1657–1668, 2023, doi: 10.1145/3611643.3613891.

[10] D. Sobania *et al.*, “Evaluating Explanations for Software Patches Generated by Large Language Models,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 14415 LNCS, pp. 147–152, 2024, doi: 10.1007/978-3-031-48796-5_12.

[11] B. Betsy, J. Chris, P. Jennifer, and M. Niall Richard, *Site Reliability Engineering: How Google Runs Production Systems*. O’Reilly Media, Inc., 2016.

[12] M. Schafer, S. Nadi, A. Eghbali, and F. Tip, “An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation,” *IEEE Trans. Softw. Eng.*, vol. 50, no. 1, pp. 85–105, 2024, doi: 10.1109/TSE.2023.3334955.

[13] C. Arora, T. Herda, and V. Homm, “Generating Test Scenarios from NL Requirements using Retrieval-Augmented LLMs: An Industrial Study,” 2024, [Online]. Available: http://arxiv.org/abs/2404.12772

[14] “ISO/IEC/IEEE International Standard - Software and systems engineering --Software testing --Part 1:General concepts,” *ISO/IEC/IEEE 29119-1:2022(E)*, pp. 1–60, 2022, doi: 10.1109/IEEESTD.2022.9698145.

[15] Z. Liu *et al.*, “Make LLM a Testing Expert: Bringing Human-like Interaction to Mobile GUI Testing via Functionality-aware Decisions,” pp. 1–13, 2024, doi: 10.1145/3597503.3639180.

[16] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, “Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing,” *ACM Comput. Surv.*, vol. 55, no. 9, pp. 1–46, 2023, doi: 10.1145/3560815.

[17] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning Internal Representations by Error Propagation,” in *Readings in Cognitive Science: A Perspective from Psychology and Artificial Intelligence*, 1985, pp. 399–421. doi: 10.1016/B978-1-4832-1446-7.50035-2.

[18] J. L. Elman, “Finding structure in time,” *Cogn. Sci.*, vol. 14, no. 2, pp. 179–211, Apr. 1990, doi: 10.1016/0364-0213(90)90002-E.

[19] M. V. M. Mahoney, “Fast text compression with neural networks,” *Proc. AAAI FLAIRS*, pp. 0–4, 2000, [Online]. Available: https://www.aaai.org/Papers/FLAIRS/2000/FLAIRS00-044.pdf

[20] A. Vaswani *et al.*, “Attention is all you need,” *Adv. Neural Inf. Process. Syst.*, vol. 2017-Decem, pp. 5999–6009, Jun. 2017, Accessed: Jul. 22, 2024. [Online]. Available: https://arxiv.org/abs/1706.03762v7

[21] A. Chowdhery *et al.*, “PaLM: Scaling Language Modeling with Pathways,” Apr. 2022, Accessed: Jul. 22, 2024. [Online]. Available: http://arxiv.org/abs/2204.02311

[22] H. Touvron *et al.*, “LLaMA: Open and Efficient Foundation Language Models,” Feb. 2023, Accessed: Jul. 22, 2024. [Online]. Available: http://arxiv.org/abs/2302.13971

[23] OpenAI *et al.*, “GPT-4 Technical Report.” 2023. [Online]. Available: http://arxiv.org/abs/2303.08774

[24] J. White *et al.*, “A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT,” 2023, [Online]. Available: http://arxiv.org/abs/2302.11382

[25] J. D. Zamfirescu-Pereira, R. Y. Wong, B. Hartmann, and Q. Yang, “Why Johnny Can’t Prompt: How Non-AI Experts Try (and Fail) to Design LLM Prompts,” *Conf. Hum. Factors Comput. Syst. - Proc.*, 2023, doi: 10.1145/3544548.3581388.

[26] H. Dang, L. Mecke, F. Lehmann, S. Goller, and D. Buschek, *How to Prompt? Opportunities and Challenges of Zero- and Few-Shot Learning for Human-AI Interaction in Creative Applications of Generative Models*, vol. 1, no. 1. Association for Computing Machinery, 2022. [Online]. Available: http://arxiv.org/abs/2209.01390

[27] J. Wei *et al.*, “Chain-of-Thought Prompting Elicits Reasoning in Large Language Models,” *Adv. Neural Inf. Process. Syst.*, vol. 35, no. NeurIPS, pp. 1–14, 2022.

[28] T. Kojima, M. Reid, and S. S. Gu, “Large Language Models are Zero-Shot Reasoners,” in *Advances in Neural Information Processing Systems 35 (NeurIPS 2022)*, 2022, no. NeurIPS.

[29] S. H. Bach *et al.*, “PromptSource: An Integrated Development Environment and Repository for Natural Language Prompts,” *Proc. Annu. Meet. Assoc. Comput. Linguist.*, pp. 93–104, 2022, doi: 10.18653/v1/2022.acl-demo.9.

[30] Microsoft, “GitHub Copilot · Your AI pair programmer,” 2022. https://github.com/features/copilot (accessed Jul. 22, 2024).

[31] C. Spiess *et al.*, “Calibration and Correctness of Language Models for Code,” Feb. 2024, Accessed: Jul. 22, 2024. [Online]. Available: http://arxiv.org/abs/2402.02047

[32] F. Liu *et al.*, “Exploring and Evaluating Hallucinations in LLM-Powered Code Generation,” Apr. 2024, Accessed: Jul. 22, 2024. [Online]. Available: http://arxiv.org/abs/2404.00971

[33] D. Nam, A. Macvean, V. Hellendoorn, B. Vasilescu, and B. Myers, “Using an LLM to Help With Code Understanding,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13. doi: 10.1145/3597503.3639187.

[34] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, “Software Testing With Large Language Models: Survey, Landscape, and Vision,” *IEEE Trans. Softw. Eng.*, vol. 50, no. 4, pp. 911–936, 2024, doi: 10.1109/TSE.2024.3368208.

[35] A. Ferrari, S. Abualhaija, and C. Arora, “Model Generation from Requirements with LLMs: an Exploratory Study,” in *2024 IEEE 32st International Requirements Engineering Conference Workshops (REW)*, 2024, pp. 291--300. [Online]. Available:

http://arxiv.org/abs/2404.06371

[36] C. Yang, J. Chen, B. Lin, J. Zhou, and Z. Wang, “Enhancing LLM-based Test Generation for Hard-to-Cover Branches via Program Analysis,” 2024, [Online]. Available: http://arxiv.org/abs/2404.04966

[37] S. Yu, C. Fang, Y. Ling, C. Wu, and Z. Chen, “LLM for Test Script Generation and Migration: Challenges, Capabilities, and Opportunities,” *IEEE Int. Conf. Softw. Qual. Reliab. Secur. QRS*, pp. 206–217, 2023, doi: 10.1109/QRS60937.2023.00029.

[38] Z. Liu *et al.*, “Fill in the Blank: Context-aware Automated Text Input Generation for Mobile GUI Testing,” *Proc. - Int. Conf. Softw. Eng.*, pp. 1355–1367, 2023, doi: 10.1109/ICSE48619.2023.00119.

[39] S. L. Shrestha and C. Csallner, “SLGPT: Using Transfer Learning to Directly Generate Simulink Model Files and Find Bugs in the Simulink Toolchain,” *ACM Int. Conf. Proceeding Ser.*, pp. 260–265, May 2021, doi: 10.1145/3463274.3463806.

[40] D. Zimmermann and A. Koziolek, “Automating GUI-based Software Testing with GPT-3,” *Proc. - 2023 IEEE 16th Int. Conf. Softw. Testing, Verif. Valid. Work. ICSTW 2023*, pp. 62–65, 2023, doi: 10.1109/ICSTW58534.2023.00022.

[41] X. Ye and G. Durrett, “The Unreliability of Explanations in Few-shot Prompting for Textual Reasoning,” *Adv. Neural Inf. Process. Syst.*, vol. 35, no. NeurIPS, pp. 1–15, 2022.

[42] ElasTest EU Project, “Fullteaching: A web application to make teaching online easy.” Universidad Rey Juan Carlos, 2017. Accessed: Aug. 10, 2023. [Online]. Available: https://github.com/elastest/full-teaching

[43] B. Garcia *et al.*, “A proposal to orchestrate test cases,” in *Proceedings - 2018 International Conference on the Quality of Information and Communications Technology, QUATIC 2018*, 2018, pp. 38–46. doi: 10.1109/QUATIC.2018.00016.

[44] P. Fuente Pérez, “FullTeaching : Aplicación Web de docencia con videoconferencia,” 2017.

[45] C. Augusto, J. Morán, C. de la Riva, and J. Tuya, “FullTeaching E2E Test Suite.” 2023. [Online]. Available: https://github.com/giis-uniovi/retorch-st-fullteaching

[46] C. Augusto, J. Moran, A. Bertolino, C. De La Riva, and J. Tuya, “Replication package for ‘Software System Testing assisted by Large Language Models: An Exploratory Study,’” *https://github.com/giis-uniovi/retorch-llm-rp*, 2024. https://github.com/giis-uniovi/retorch-llm-rp (accessed Jul. 22, 2024).

[47] Peter Krensky *et al.*, “Magic Quadrant for Data Science and Machine Learning Platforms,” 2020. [Online]. Available: https://qads.com.br/data-analytics/pdfs/Gartner%0A2018.pdf

[48] OpenAI, “Cheat Sheet: Mastering Temperature and Top_p in ChatGPT API - API - OpenAI Developer Forum,” *OpenAI Documentation*, 2023. https://community.openai.com/t/cheat-sheet-mastering-temperature-and-top-p-in-chatgpt-api/172683 (accessed Jul. 23, 2024).

[49] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*, vol. 9783642290. 2012. doi: 10.1007/978-3-642-29044-2.