On the Soundness and Consistency of LLM Agents for Executing Test Cases Written in Natural Language [a]

Sébastien Salva[1,2] and Redha Taguelmimt[1,2]
[1] *IUT CA, Clermont Auvergne University, UCA, Aubière, France*
[1] *LIMOS - UMR CNRS 6158, Aubière, France*
sebastien.salva@uca.fr, redha.taguelmimt@uca.fr

Keywords:
GUI Application, Natural Language Test Case, Soundness, Execution Consistency, LLM agents.

Abstract:
The use of natural language (NL) test cases for validating graphical user interface (GUI) applications is emerging as a promising alternative to manually written executable test scripts, which are costly to develop and difficult to maintain. Recent advances in large language models (LLMs) have opened the possibility of the direct execution of NL test cases by LLM agents. This paper explores this direction, with particular attention to NL test case unsoundness and to the consistency of test case execution. NL test cases are inherently unsound because ambiguous instructions or unpredictable agent behaviour can produce false failures. Furthermore, repeated executions of the same NL test case may lead to inconsistent outcomes, undermining test reliability. To address these challenges, we propose an algorithm for executing NL test cases with specialised agents and guardrail mechanisms that dynamically verify the test step executions. We introduce measures to evaluate the capabilities of LLM agents in test execution and a measure to estimate NL test case execution consistency. We also propose a definition of weak unsoundness capturing contexts where rare incorrect verdicts are tolerable. Our experimental evaluation with eight publicly available LLMs demonstrates the potential of LLM agents for GUI testing. Our experiments show that Meta Llama 3.3 70B demonstrates good capabilities in NL test case execution with respect to the industrial quality levels 3-Sigma (mean accuracies greater than 98%), with high execution consistency.

1 INTRODUCTION

Scenario

Open the website https://www.uca.fr/en, search for a page having news related to the **ARTEMIS** project.

Figure 1: Scenario to search for ARTEMIS project news

In industrial settings, the development and test of GUI applications often involves the use of scenarios to express user requirements, which are then refined into test cases written in natural language and subsequently into concrete test case scripts. As an illustrative example, Figure 1 presents a simple scenario in which a user searches for pages related to a project on the website uca.fr/en. From this scenario, natural language (NL) test cases are derived; Figure 2 shows an example for the previous scenario. The NL test cases are then implemented as executable test scripts, also shown in Figure 3.

Test Case

- Open the website https://www.uca.fr/en
- Click on `European University`
- Click on `ALL NEWS`
- Assert that the page has links
- Assert that there is a link with the term **ARTEMIS**

Figure 2: Step-by-step NL test case for verifying the presence of a link containing the term 'ARTEMIS'

[a]Research supported by the Industrial Chair on Reliable and Confident Use of LLMs (https://uca-fondation.fr/les-chaires/) and MIAI Cluster, France 2030 (ANR-23-IACL-0006)

Selenium Test

```
@Test
public void testSimple(){
WebDriver driver = new FirefoxDriver();
 driver.get("https://www.uca.fr/en");
 WebElement element = driver.findElement(
 By.xpath("/html/body/header/div[3]/div[2]/
      nav/ul[1]/li[5]/a"));
 element.click();
 element = driver.findElement(
 By.xpath("/html/body/main/div[1]/div/div
      [1]/div[2]/div/div[3]/div/div/div/div
      [1]/a"));
 element.click();
 List<WebElement> links = driver.
      findElements(By.xpath("//a[@href]"));
 assertTrue(links.stream().anyMatch(
 l -> l.getText().contains("ARTEMIS")));
 driver.quit();}
```

Figure 3: Automated Selenium test implementing the test case of Figure 2

Writing concrete test cases such as the one in Figure 3 is time-consuming, and the resulting source code can be difficult to maintain, particularly when UI elements change over time, e.g., the "/html/body/..." elements in the figure. As a result, functional tests are often neglected in practice. Artificial intelligence (AI) has the potential to reduce the effort involved in test development. Currently, two promising use cases warrant particular attention and evaluation: 1) the test case generation with Large Language Models (LLMs) from high-level scenarios or NL test cases. This approach can significantly reduce the manual effort required, although evaluating the correctness of the generated source code remains a challenging and time-consuming task [Augusto et al., 2024];
2) the direct execution of NL test cases by means of agents powered by LLMs. Such LLM agents can indeed simulate user interactions with the GUI [Yoon et al., 2024, Liu et al., 2024], potentially allowing the complete execution of NL test cases.

This paper focuses on the second possibility by exploring how LLM agents can test GUI applications. Specifically, we address the following questions: can LLM agents be effective for testing GUI applications with NL test cases? How does the use of these agents affect the test case soundness and the reproducibility of their execution (i.e. execution consistency)?

It must be stated unequivocally that NL test cases are inherently unsound, i.e., they may reject an existing conformant implementation. Unsoundness primarily stems from ambiguity in NL actions, whose interpretations may lead to a fail verdict. Moreover, the execution of NL test cases by AI agents introduces additional risks. AI agents may behave unpredictably, or even hallucinate steps that are not specified. Additionally, the execution of NL test cases by AI agents introduces inconsistencies because different executions of the same test case may yield varying behaviours depending on factors such as prompt formulation or agent capabilities. These inconsistencies undermine test case execution consistency.

1.1 Our Contributions

Our contributions consist of three interrelated components.

A guarded execution algorithm for NL test cases. We propose a NL test case execution algorithm that integrates guardrail mechanisms. These mechanisms are based on rule-based checks (e.g., verifying GUI element visibility and validating preconditions) or validation prompts that ensure each proposed action remains consistent with the intended test behavior. During test execution, the algorithm dynamically completes and controls the NL test steps ensuring that each step is carried out as intended. It relies on a set of specialised LLM agents that independently handle key subtasks: navigating the GUI, verifying that the interface is ready for the next action, and evaluating assertions against the current GUI state. To the best of our knowledge, this is the first algorithm to integrate guardrail mechanisms and specialized LLM agents to control the execution of NL test cases.

A formalization of weak unsoundness in NL test execution. Given that full soundness is intractable, we identify the contexts in which NL test case unsoundness is acceptable. We introduce the notion of weak unsoundness that captures situations in which the probability of an incorrect fail verdict remains below a small predefined tolerance threshold. We formally characterise the contexts in which NL test cases can be reliably executed by LLM agents under weak unsoundness.

Agent effectiveness and execution consistency measures with empirical evaluation.

To assess the capabilities of LLM agents in performing actions, we define three measures. In addition, we introduce another measure to estimate the execution consistency of NL test cases with respect to agent capabilities—that is, the ability of our algorithm to produce stable and repeatable test outcomes. Furthermore, this paper presents experiments evaluating the capabilities of eight publicly available LLMs, which are deployable on local servers, in executing NL test cases. Our results show that while some LLMs are effective, further work is required to enhance their capabilities.

2 RELATED WORK

AI has been applied to various software engineering activities, with steadily increasing adoption over the past decade. Some surveys covered the use of machine learning [Allamanis et al., 2018] or LLMs for software engineering [Hou et al., 2024].

AI and more specifically LLM-based approaches have also been proposed for testing. ChatUniTest [Chen et al., 2024] is an LLM-based automated unit test generation framework. It incorporates an adaptive focal context mechanism to encompass valuable context in prompts and adheres to a generation-validation-repair mechanism to rectify errors in generated unit tests. The evaluation shows that ChatUniTest outperforms TestSpark and EvoSuite in half of the evaluated projects, achieving the highest overall line coverage. Fuzz testing, which aims at providing a program with unexpected or random inputs to identify bugs, is also improved with AI. For example, LLMFuzz [Deng et al., 2023] is a testing tool that calls LLMs to generate inputs for fuzzing Deep Learning libraries.

Closer to our work, a few algorithms that extend Android crawlers were recently proposed to detect bugs. Instead of using meta-heuristics [Salva and Zafimiharisoa, 2014], they call LLMs or LLM agents to better cover a mobile application. DroidAgent [Yoon et al., 2024] is a multi-agent tool for autonomous test generation for Android applications. It gathers three agents to interact, observe, and evaluate task success. To optimise the application coverage, the agents have three types of memory: short-term memory (GUI state), long-term memory (history of performed tasks) and spatial memory to record observations made after interacting with specific widgets. It outperforms existing state-of-the-art tools by achieving 10% higher coverage than the baseline. GPTDroid [Liu et al., 2024] is another tool for Android testing that covers an application to detect bugs. It extracts the GUI context, uses a mechanism that records past testing interactions (activities, widgets, operations), encodes them into prompt questions for the LLM, decodes the LLM answer into actionable operations to execute on the application, and iterates the whole process. On 93 popular applications, GPTDroid achieves 32% higher coverage than the baseline. These approaches dynamically explore the application's navigation tree using LLM agents to find bugs. Since this process operates without predefined test cases, it lacks repeatability, making consistent test execution infeasible. In contrast, our work focuses on the execution of conformance test cases and on test case execution consistency.

The exploratory study presented in [Augusto et al., 2024] investigates how LLMs can support system test development—that is, the generation of NL test cases as well as the creation of concrete test cases through specialised LLM prompts. Their study, conducted on one subject, shows that when user requirements are fully given, LLMs are able to provide NL test cases that cover most of them. However, generating concrete test cases is more challenging, as these require at least 30% of modifications before becoming executable.

In contrast, our work assumes that NL test cases are already available and investigates the alternative possibility of executing them directly on a GUI application under test using LLM agents. Futhermore, none of the surveyed approaches analyse the soundness or execution consistency of NL test cases, nor do they provide a formal framework to quantify when an NL-driven execution may yield incorrect verdicts. This gap directly motivates our proposal for a principled execution model and a definition of weak unsoundness for NL test cases.

3 NL TEST CASE EXECUTION

3.1 Notation and Assumptions

We assume having a black box GUI application under test, denoted AUT, where GUIs expose either a structured representation (e.g., DOM or navigation tree) or screenshots that can be captured during interaction. In this paper, we focus

on web applications because of the availability of browser automation tools, but the underlying algorithm is compatible with mobile applications with appropriate adapters.

In testing theory, specifications, implementations, test cases and implementation relations are expressed or formulated using formal models. We adopt in this paper the classical model of *input output labelled transition systems (IOLTS)*. We assume that the application under test, AUT, could in principle be modelled by a specification. More precisely, we posit that there exists a deterministic IOLTS $S = \langle Q^S, L^S \cup \mathcal{T}, \rightarrow^S, q_0^S \rangle$ that accurately captures the intended behaviour of the application, where Q^S denotes the set of states, $L^S = L_I^S \cup L_O^S$ the set of input and output actions (prefixed by "?" and "!" respectively), $\mathcal{T}$ the set of internal actions, $\rightarrow^S$ the transition relation, and q_0^S the initial state. While we consider that this specification is not necessarily available to the tester, assuming its existence allows us to relate the execution of NL test cases to classical conformance testing theory and to reason formally about test case properties. In the present work, inputs correspond to navigation actions, such as clicking on "Sign in", and outputs correspond to the resulting GUIs. The specified behaviours of the application can be extracted from the traces of S, denoted $traces(S)$, which gathers all the sequences of observable inputs and outputs obtained by covering the paths of S (definitions in [Salva and Taguelmimt, 2025]).

A NL test case is a sequence of actions $a_1 \dots a_k \, \mathcal{A}_{k+1} \dots \mathcal{A}_l$, where $a_1, \dots, a_k \in L_I^S$ are navigation actions and $\mathcal{A}_{k+1}, \dots, \mathcal{A}_l \in \mathcal{T}$ are assertions treated as internal actions. Each assertion $\mathcal{A}$ may be a simple predicate or a composite expression formed through conjunctions and disjunctions of simpler assertions. With this formulation, navigation actions and assertions cannot be interleaved, meaning that assertions are interpreted as postconditions. However, interleaved navigation actions and assertions can be modeled in our framework as a sequence of smaller NL test cases, each with its own prefix of actions followed by assertions. This provides a practical decomposition strategy that allows to keep our test case execution algorithm more readable. A test case $tc = a_1 \dots a_k \, \mathcal{A}_{k+1} \dots \mathcal{A}_l$ is consistent with the specification S by assuming the existence of a trace $?a_1 \, !g_1 \dots ?a_k \, !g_k \in traces(S)$, and if each assertion $\mathcal{A}_i (k+1 \leq i \leq l)$ evaluates to true on the GUI state g_k. This condition ensures that the actions and assertions encoded in a NL test case correspond to behaviours permitted by the (unknown) specification and thus serve as a faithful abstraction of the intended system behaviour. The trace structure reflects a deliberate modelling assumption aligned with the behaviour of web applications: after each user interaction (input), the application produces an observable GUI state (output). In other words, we assume that interactions are reactive and yield observable effects.

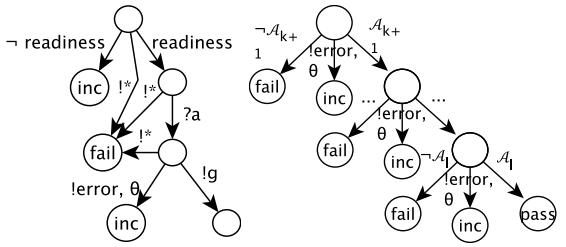


Figure 4: Illustration of the test case execution algorithm. (a) Navigation actions are injected with readiness. For readability, "!*" represents any other output. (b) Assertions are incrementally evaluated to determine the final verdict.

3.2 NL Test Case Execution Algorithm

The experimentation of AUT with a NL test case tc using LLM agents introduces several challenges, including the design of agents, their control to ensure that each action of the NL test case is correctly executed, or the need to get consistent test verdicts across different test executions.

With this in mind, we propose a NL test case execution algorithm that constructs the IOLTS $tc|AUT =< Q, L \cup \mathcal{T} \cup \{\theta\}, \rightarrow, q_0 >$ modelling the synchronous execution of tc on AUT. θ is a specific symbol modelling quiescence, i.e., the fact that AUT has produced no output. This model enables us to formally relate NL test case execution to classical conformance testing theory. In particular, it serves as the basis for defining the notion of weak unsoundness for NL test cases, by characterising the conditions under which an executed test may incorrectly reject a conformant AUT (Section 5).

Algorithm 1 adds to the original NL test case the notion of inputs, i.e. navigation actions applied or sent to AUT, and outputs, i.e. GUIs initiated by AUT. It also introduces an internal action, denoted *readiness*, which is systematically injected for every navigation action, as illustrated in Figure 4a. This action is said to be

internal in the sense that it is not observable during execution. *readiness* checks whether the next navigation action is executable, i.e., whether the necessary UI elements are present on the interface to perform the action. By verifying GUI readiness (visibility, interactability, stability of the DOM) before interacting with the GUI, Algorithm 1 explicitly limits the conditions under which an incorrect fail verdict can occur. In other terms, *readiness* operationally lowers the probability of false fail outcomes. *readiness* also contributes to evaluating the consistency and reliability of the test case execution (Section 4).

Algorithm 1: NL test case execution

Input : NL Test case $tc = a_1 \dots a_k \, \mathcal{A}_{k+1} \dots \mathcal{A}_l$
Output: Test verdict, IOLTS $tc|AUT =< Q, L \cup \mathcal{T} \cup \{\theta\}, \rightarrow, q_0 >$

1 $i := 1$;
2 **while** $i \leq k \ AND \ \neg \ stop$ **do**
3 //**Observe** AUT;
4 Add $q_{i-1} \xrightarrow{!*} fail$ if output observed; Return fail; stop;
5 //**Check Readiness**;
6 Add $q_{i-1} \xrightarrow{readiness} q_{i,1}$ if $readiness(a_i)$ true;
7 Add $q_{i-1} \xrightarrow{\neg readiness} inc$ if $readiness(a_i)$ false; Return inconclusive; stop;
8 //**Observe** AUT;
9 Add $q_{i,1} \xrightarrow{!*} fail$ if output observed; Return fail; stop;
10 //**Give** a_i **to** AUT;
11 Execute a_i;
12 Add $q_{i,1} \xrightarrow{?a_i} q_{i,2}$;
13 //**Observe** AUT;
14 Add $q_{i,2} \xrightarrow{!g} q_i$ if GUI g observed;
15 Add $q_{i,2} \xrightarrow{!error} inc$ if $error$ observed; Return inconclusive; stop;
16 Add $q_{i,2} \xrightarrow{\theta} inc$ if quiescence is observed; Return inconclusive; stop;
17 Add $q_{i,2} \xrightarrow{!*} fail$ if any other output observed; Return fail; stop;
18 i++;
19 //**Assertions on last GUI**;
20 **while** $k + 1 \leq i \leq l \ AND \ \neg \ stop$ **do**
21 Add $q_{i-1} \xrightarrow{\mathcal{A}_i} q_i$ if $\mathcal{A}_i$ true on $!g$ AND $i \neq l$;
22 Add $q_{i-1} \xrightarrow{\mathcal{A}_i} pass$ if $\mathcal{A}_i$ true on $!g$ AND $i = l$; Return pass; stop;
23 Add $q_{i-1} \xrightarrow{\neg\mathcal{A}_i} fail$ if $\mathcal{A}_i$ false on $!g$; Return fail; stop;
24 Add $q_{i-1} \xrightarrow{!error} inc$ if error observed; Return inconclusive; stop;
25 Add $q_{i-1} \xrightarrow{\theta} inc$ if quiescence observed; Return inconclusive; stop;
26 i++;

Algorithm 1 starts by covering every navigation action a_i in $a_1 \dots a_k$ (lines 3-18). It builds $tc|AUT$ as illustrated in Figure 4a. If $readiness(a_i)$ evaluates to false on the GUI g, or if quiescence or an error is observed, the test case execution ends with an inconclusive verdict. The latter serves as a protective verdict that prevents the system from being incorrectly rejected when the execution context is unreliable on account of LLM agents or of environment issues. If unexpected output actions are observed, the test case execution ends and a fail verdict is returned. Otherwise, the algorithm continues with the next action. When the algorithm reaches the assertions $\mathcal{A}_{k+1} \dots \mathcal{A}_l$ (lines 20-26), it incrementally evaluates them, as illustrated in Figure 4b. If an assertion evaluates to false, then a fail verdict is returned. The inconclusive verdict is also reached in case an error is triggered by an agent or quiescence is observed. Otherwise, a pass verdict is given. We now detail how every action (navigation action, assertion, readiness) are performed:

3.3 Navigation Action Execution

Navigation actions are carried out by an agent that integrates an LLM for reasoning and decision-making, a memory module, and a function invocation interface.

The memory module allows to store the current execution state (finished, error), step queues, intermediate results, and also incorporates an historical context, which potentially allows the agent to revisit previously accessed pages and make informed decisions based on past interactions. The function invocation interface enables the agent to interact programmatically with GUI content or to interpret visual information extracted from screenshots.

The agent is invoked using prompts we wrote and structured following the patterns proposed in [White et al., 2023]. We here considered: 1. the "Fact Checklist" pattern that asks the agent to output a list of facts included in the final answer, thereby enhancing transparency on the results provided; 2. the "Template" pattern that enforces an output format in JSON, which includes fields such as extracted facts, results, and a boolean indicator of task completion status; 3. The "Recipe" pattern, which builds a complete sequence of steps given some partial "ingredients" to achieve the given task. The prompt is given in [Salva and Taguelmimt, 2025].

3.4 Readiness Actions Execution

$readiness(a)$ checks whether the current GUI is ready for the execution of the upcoming navigation action a, i.e., whether the requisite UI elements are present. $readiness(a)$ is defined by the predicate logic formula $readiness\_strict(a) \vee (\neg readiness\_strict(a) \implies readiness\_w\_agent(a))$, which evaluates to a boolean value. Here, $readiness\_strict$ is a boolean procedure that performs the following steps:

1. translation of a into a predicate;
2. evaluation of the predicate with the formula $\bigwedge_1^k \phi_i$, where ϕ_i are predicates that express actions performed on the GUI and constraints.

For instance, consider the navigation action "Click the link 'Sign in'". A sub-formula ϕ written to evaluate whether this action can be performed on the GUI can be: $\phi : \forall x, \text{Click}(x) \implies \text{In}(x, content)$ with the predicates $\text{Click}(x) \implies$ "Element x is clicked" and $\text{In}(x) \implies$ "Element x is in the GUI content". Table 1 lists the sub-formula we designed to perform the evaluation of $readiness\_strict$.

Table 1: Examples of formulas for evaluating GUI readiness.

Action	Formula
click(x)	$\forall x, \text{Click}(x) \implies \text{In}(x, \text{content}) \wedge (\text{Type}(x, \text{"link"}) \vee (\text{Type}(x, \text{"button"}))$
select(x, v)	$\forall x, \text{Select}(x, v) \implies \text{In}(x, \text{content}) \wedge \text{Type}(x, \text{"list"}) \wedge \text{In}(v, \text{option(x)})$
check(x)	$\forall x, \text{Check}(x) \implies \text{In}(x, \text{content}) \wedge \text{Type}(x, \text{"checkbox"}) \wedge \neg\text{Checked}(x)$
fill(x, v)	$\forall x, \text{Fill}(x, v) \implies In(x, \text{content}) \wedge \text{Type}(x, \text{"input"})$

Using $readiness\_strict(a)$ offers the advantage of providing an unambiguous guarantee that the subsequent action can be executed directly on the GUI. However, writing a fully exhaustive formula is inherently challenging, given the dynamic and evolving nature of GUI interactions. This is why we use the second boolean procedure, $readiness\_w\_agent(a)$. An agent is here requested to assess GUI readiness by extracting all the UI elements of the GUI and by checking if the action a can be executed with these elements. We request the agent with a prompt we structured with the following patterns [White et al., 2023]: "Fact Check List" and "Template", similar to the prompt written to perform navigation actions. We also employed the "Chain of Thought" prompt engineering technique [Wei et al., 2022] to enable the reasoning capabilities of the agent by producing intermediate steps. The prompt is given in [Salva and Taguelmimt, 2025].

3.5 Assertion Evaluation

The most reliable way to ensure the validity of an assertion $\mathcal{A}$ is to express it using a strict logical formula. However, automatically translating textual assertions into such formal representations is often a long and non-trivial activity. To address this, we adopt a strategy analogous to the one used with $readiness$. An assertion $\mathcal{A}$ is evaluated by the logical formula $assert\_strict(\mathcal{A}) \vee (\neg assert\_strict(\mathcal{A}) \implies assert\_w\_agent(\mathcal{A}))$. Here, $assert\_strict$ refers to a boolean procedure that performs the following steps: 1. split of $\mathcal{A}$ into individual elementary assertions separated by operators; 2. translation of the elementary assertions into predicates; 3. evaluation of every predicate by means of the formula $\bigwedge_{i=1}^k \phi_i$; 4. evaluation of the final boolean according to the operators in $\mathcal{A}$. The sub-formula ϕ_i expresses simple but usual generic assertions considered in GUI testing. For example, "Assert that 'x' is present" is expressed by $\phi : \forall x, Present(x) \implies In(x, page)$. Table 2 lists several sub-formula.

Table 2: Examples of assertions expressed with formulas.

Assertion	Formula
'x' is present	$\forall x, \text{IsPresent}(x) \implies \text{In}(x, \text{content})$
'x' is not present	$\forall x, \neg\text{IsPresent}(x) \implies \neg\text{In}(x, \text{content})$
'x' is checked	$\forall x, \text{IsChecked}(x) \implies \text{In}(x, \text{content}) \wedge \text{Checked}(x)$
'x' is visible	$\forall x, \text{IsVisible}(x) \implies \text{In}(x, \text{content}) \wedge \text{Visible}(x)$

The predicate $assert\_w\_agent(\mathcal{A})$ expresses the invocation of an agent to evaluate the assertion. This allows the use of more complex, but potentially more ambiguous, assertions such as "Assert that the page has more than three links". The agent is requested with another prompt also constructed with the patterns "Fact Check List", "Template" and "Recipe". The latter is specifically used to force the agent to focus on the UI elements relevant to the assertion, before evaluating them and giving a final verdict. It also uses "Chain of Thought" to explicitly require the generation of intermediate steps before generating the final response. The prompt is given in [Salva and Taguelmimt, 2025].

4 TEST CASE EXECUTION CONSISTENCY

We introduce a measure for quantifying the estimated consistency of test case execution w.r.t. the selected LLM agents, $agent_{nav}$ (navigation), $agent_{readiness}$ (GUI readiness evaluation), and $agent_{assert}$ (assertion evaluation). For each test case tc, the goal is to estimate the degree to which its verdicts remain stable over repeated executions.

A naive way to measure consistency is to run a test case multiple times and check whether the verdicts and observed behaviours remain unchanged across runs. This approach is time consuming. Instead, we exploit the fact that the consistency of the test case execution depends on the abilities of each agent to perform its tasks in a predictable way. Hence, we initially assess the ability of every agent to perform its tasks by computing the standard deviation on its binary results, which measures the variation of the values of a variable. The standard deviation of the results produced by an agent is given by $0 \leq \sigma(agent) = \sqrt{p(1-p)} \leq 0.5$, with p the probability of success for doing its tasks. The closer $\sigma(agent)$ is to 0, the more stable its execution is across multiple runs. To compute $\sigma(agent)$, we use a predefined set of test cases in which the expected outcomes (success/failure) of actions are known in advance. Each specialised agent is evaluated on this set, and its observed successes and failures provide p. This procedure is described in greater detail in Section 6 in relation to RQ1.

To measure the consistency of execution of a test case, we define three measures.

First, $s_r(a)$ measures the consistency of $readiness(a)$. Recall that $readiness(a)$ is either performed by $readiness\_strict(a)$ or $readiness\_w\_agent(a)$. $readiness\_strict$ is expressed by a formula and is consistent. When the formula evaluates to false, we use $readiness\_w\_agent(a)$, which itself calls $agent_{readiness}$, whose execution consistency is related to the standard deviation of $agent_{readiness}$. s_r is hence defined as:

$$0 \leq s_r(a) \leq 1 =_{def} \begin{cases} 1 \text{ if } readiness\_strict(a) = 1 \\ 1 - 2\sigma(agent_{readiness}) \\ \text{otherwise} \end{cases}$$

Second, $s_n(a)$ measures the consistency of performing a navigation action a. This execution is exclusively made by $agent_{nav}$. Consequently, s_n is defined as:

$$0 \leq s_n(a) \leq 1 =_{def} 1 - 2\sigma(agent_{nav})$$

Third, $s_a(\mathcal{A})$ measures the consistency of the evaluation of $\mathcal{A}$. Similarly to $readiness$, this action is either performed by $assert\_strict(\mathcal{A})$ or $assert\_w\_agent(\mathcal{A})$. s_a is hence defined as:

$$0 \leq s_a(\mathcal{A}) \leq 1 =_{def} \begin{cases} 1 \text{ if } assert\_strict(\mathcal{A}) = 1 \\ 1 - 2\sigma(agent_{assert}) \\ \text{otherwise} \end{cases}$$

The consistency of a test case $tc = a_1 \dots a_k \mathcal{A}_{k+1} \dots \mathcal{A}_l$ is defined as the mean of the consistency scores of each action:

$0 \leq consistency(tc) \leq 1 =_{def}$

$$x \frac{1}{l} (\sum_{1}^{k} s_e(a_i) s_n(a_i) + \sum_{k+1}^{l} s_a(\mathcal{A}_i))$$

5 TEST CASE UNSOUNDNESS

A test case is said unsound if there exists at least one conformant AUT that is rejected by the test case [Jard et al., 2000]. This section aims at delimiting the impact of agent use to execute NL test cases on unsoundness.

This property is typically defined with respect to formal models and requires a specification and an implementation relation. In our context, we supposed that there exist a specification S and that a NL test case $tc = a_1 \dots a_k \mathcal{A}_{k+1} \dots \mathcal{A}_l$ is extracted from S such that $\exists ?a_1 ! g_1 \dots ?a_k ! g_k \in traces(S)$ and $\mathcal{A}_i (k+1 \leq i \leq l)$ are true on the GUI g_k. Conformance testing also relies on implementation relations that characterise how an implementation adheres to its formal specification. Here, we consider the $ioco$ relation [Tretmans, 1996]. Under $ioco$, AUT conforms to a specification S if every action sequence derivable from S yields, when executed on AUT, only outputs anticipated by S. Here, we do not have formally modeled test cases, but Algorithm 1 dynamically builds the IOLTS $tc|AUT$ expressing the execution of tc on AUT and adds the notions of inputs, outputs, along with internal actions that lead to terminal states labelled by verdicts.

The execution of tc is unsound as our LLM agents may hallucinate and return wrong responses when they interact with GUIs and evaluate actions. Even when LLM agents operate with high accuracy, the tc execution remains unsound because there still exists a small probability of producing false verdicts. In many practical contexts, however, this residual probability may be

deemed acceptable compared to other sources of uncertainty, such as environmental disturbances, which can similarly affect test execution.

From this observation, we introduce the notion of weak unsoundness, defined with respect to rare contexts in which a test case may be executed. In general terms, a test case is said weakly unsound if there exists a conformant implementation that is rejected by the test case only under rare circumstances. We make that formal through the following definition. Let Ω denote the set of all possible contexts, and Ω_r denote a low-probability subset of Ω representing exceptional conditions.

Definition 1 (Weak Unsoundness)**.** *Let S be an IOLTS $< Q^S, L^S \cup \mathcal{T}, \rightarrow^S, q_0^S >$, $tc = a_1 \dots a_k \mathcal{A}_{k+1} \dots \mathcal{A}_l$ be a NL test case such that $\exists ?a_1!g_1 \dots ?a_k!g_k \in traces(S)$ and $\mathcal{A}_j (k+1 \leq j \leq l)$ are true on g_k, and $tc|AUT$ be a deterministic IOLTS given by Algorithm 1.*

- *AUT passes tc iff $\forall \sigma \in (L^S)^*, tc|AUT \overset{\sigma}{\not\Rightarrow} fail$*
- *tc is weakly unsound w.r.t. S for ioco iff $\forall AUT, AUT$ ioco S, $\forall c \in \Omega \setminus \Omega_r, AUT$ passes $tc \wedge \exists AUT, AUT$ ioco $S, \exists c \in \Omega_r, \neg(AUT$ passes $tc)$*

Within our context, weak unsoundness should reflect a practical tolerance for minor uncertainties inherent to the use of agents to perform their tasks for the execution of NL test cases. We abstract a context c with the form $\langle \sigma(agent_{readiness}), \sigma(agent_{nav}), \sigma(agent_{assert}) \rangle$ expressing the deviations of our agents related to their stochastic and configurable aspects. This proposition provides a sufficient condition for weak unsoundness by relating the average reliability of the agents to a user-chosen tolerance bound α.

Proposition 2. *If $\langle \sigma(agent_{readiness}) < \alpha, \sigma(agent_{nav}) < \alpha, \sigma(agent_{assert}) < \alpha \rangle \in \Omega_r$ then tc is weakly unsound.*

In the definition, the fact that there exists $?a_1!g_1 \dots ?a_k!g_k \in traces(S)$, and $\mathcal{A}_j (k+1 \leq j \leq l)$ are true on g_k implicitly supposes that the test case actions are unambiguous and can be executed with success on a conformant AUT. Unfortunately, it is inherently difficult to establish whether an action or assertion expressed in natural language is free of ambiguity. This issue can be addressed by exclusively writing NL test cases with actions and assertions that can be evaluated with $readiness_{strict}$ and $assert_{strict}$:

Proposition 3. *If tc is expressed only with navigation actions and assertions that can be evaluated with readiness_strict and assert_strict, and $\langle \sigma(agent_{readiness}), \sigma(agent_{nav}) < \alpha, \sigma(agent_{assert}) \rangle \in \Omega_r$, then tc is weakly unsound.*

Proofs of these propositions are given in [Salva and Taguelmimt, 2025].We propose a quantitative characterisation of the user-chosen error tolerance α in terms of Sigma levels (Six-Sigma method [Godfrey, 2014]). Specifically, we require that agent performance goes above 3-Sigma ($p_3 = 0.9332$, $3\sigma = 0.2496$), knowing that 3-Sigma is the minimum acceptable level in many industries. Our experimental results indicate that, at present, Propositions 2 and 3 hold with a few LLMs, which can be deployed on local servers, with $\alpha = 3\sigma$. We believe that ongoing advancements in both LLM and agent technologies will soon render Propositions 2 and 3 attainable with more small language models.

6 EXPERIMENTAL RESULTS

We investigated the capabilities of our algorithms through the following questions:

- RQ1: LLM agent Effectiveness: How effectively can LLM agents perform navigation actions and evaluate readiness actions and assertions?
- RQ2: Predicted vs Observed Consistency: How do the estimated consistency measures $consis\ tency(tc)$ compare with the observed test case execution consistency?

To answer these questions, we developed 2 prototype tools composed of three agents. Two agents evaluate readiness actions and assertions, as described in Section 3. The last agent was built on top of the Stagehand [1] framework, which automates browser interactions with natural language and code by means of LLMs:

- EvalAgent is a tool specialised in computing the 3 standard deviation scores $\sigma(agent_{readiness})$, $\sigma(agent_{nav})$, and $\sigma(agent_{assert})$; It takes an LLM, a test suite and runs it N times to compute standard deviations;
- NLTestRunner implements Algorithm 1.

[1] https://www.stagehand.dev/

We conducted our experiments using eight locally deployable LLMs to ensure reproducibility, with model sizes ranging from 3B to 70B parameters. We selected LLMs that support tool calling, which is a required feature to allow GUI interaction. The LLM inference was carried out using Ollama[2], installed on a local server equipped with an NVIDIA L40 GPU (48 GB memory).

We also designed 4 evaluation test suites, whose descriptions are summarised in Table 3 for 6 web sites (Google Gruyere, UCA, ARTEMIS, personal website, Water-Management-System Opencart). These test suites, the tools along with the web site source codes are available in https://github.com/FondationUCA-Chair-LLM/NL-test-case-runner.

6.1 RQ1: LLM agent Effectiveness

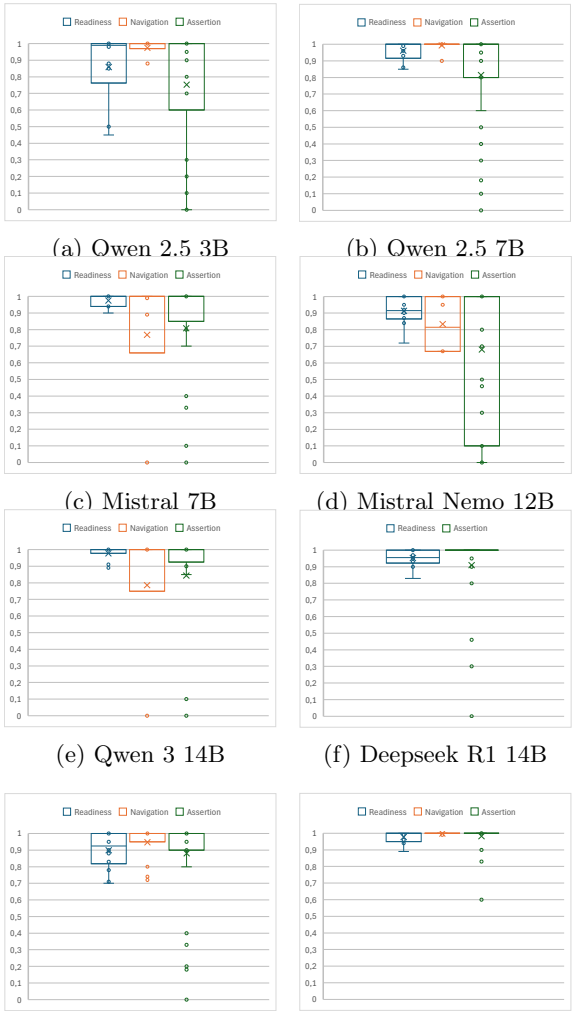
Setup: To address this question, we considered these 8 LLMs: Qwen2.5:3B, Qwen2.5 7B, Mistral 7B, Mistral Nemo 12B, Deepseek-R1 14B, Qwen3:14B, Mistral Devstral 24B, and Llama3.3:70B. We designed two special test suites TestG and TestA for this evaluation. TestG includes 16 NL test cases, each consisting of 4 to 15 steps and 1 to 4 assertions. TestA is a test suite specialised in assessing the capabilities of agents to evaluate assertions, which includes 29 test cases. Each test case is supplemented with a boolean tabular specifying the expected outcome of every step. To broadly evaluate agent capabilities, the test cases include both positive assertions (where a Pass verdict is expected) and negative assertions (where a Fail verdict is expected).

For each LLM used by the 3 LLM agents, and for each NL test case, we evaluated the accuracy of each step category (readiness, navigation, and assertion) across 20 repeated runs. For each category, we computed both the (mean) accuracy and the standard deviation over these runs.

Results: Figure 5 presents box-plots of the distributions of mean accuracy across the NL test cases. The plots also show quartiles and mean values (indicated with crosses) of performing navigation actions, returning expected readiness, and returning expected assertion verdicts. Table 4 lists the standard deviations derived from these experiments.

Focusing on an easily interpretable measure, the mean accuracy, we observe a variability in performance across LLMs and tasks and identify three main groups of LLMs:

[2]https://ollama.com/



(a) Qwen 2.5 3B (b) Qwen 2.5 7B

(c) Mistral 7B (d) Mistral Nemo 12B

(e) Qwen 3 14B (f) Deepseek R1 14B

(g) Mistral Devstral 24B (h) Llama 3.3 70B

Figure 5: Ability of LLM agent to perform readiness actions, navigation actions and assertions measured as mean accuracies with TestG and TestA over a batch of 20 runs

- Mean accuracies $> 93.32\%$ (level 3σ): considering all test case step categories, only one model showed acceptable capabilities in running NL test cases. Llama 3.3 70B achieved the best performance in our experiments, with mean accuracies greater than or equal to 98%. However, the box plot reveals some outliers in assertion evaluations;
- Mean accuracies $\geq$ 80%: Qwen 2.5 7B, DeepSeek R1, and Devstral 24B fall into this group. The box plot shows that Qwen 2.5 7B performs well in executing readiness and navigation actions, but yields mixed results when evaluating assertions. DeepSeek R1 is unable to execute navigation actions despite being a model that supports tool calling. It performs

Table 3: Summary and details of the evaluation test suites.

Test Suite	# test cases	# steps total	# assertions total	# fail test cases	# web sites covered
TestG	16	116	19	6	3
TestA	29	58	29	8	3
TestO	10	49	12	0	1
TestM	10	82	11	0	1

Table 4: Standard deviation results for readiness, navigation, and assertion actions.

LLM	Readiness Std. dev.	Navigation Std. dev.	Assertion Std. dev.
Qwen 2.5 3B	0.348	0.158	0.431
Qwen 2.5 7B	0.192	0.084	0.388
Mistral 7B	0.154	0.421	0.393
Mistral Nemo 12B	0.280	0.372	0.466
Qwen 3 14B	0.147	0.410	0.363
Deepseek R1 14B	0.213	/	0.286
Mistral Devstral 24B	0.300	0.224	0.324
Llama 3.3 70B	0.149	0.038	0.132

well in evaluating readiness and assertions but it is less effective than Llama 3.3 70B.

- Mean Accuracies $< 80\%$: the remaining four models can execute some NL test cases correctly, but frequently fail on navigation actions. Qwen 2.5 3B seems to perform well for GUI interaction but poorly on assertions, revealing a sort of bias in our experiments. We observed that failed assertions may sometimes come from incorrectly executed navigation actions. Some LLM agent can return a success in performing a navigation action despite an incorrect real execution. Since the GUI state then diverges from the expected one, assertions fail. Our current NL test case dataset, TestG, does not allow precise detection of navigation errors. This is why we also use TestA in these experiments.

For the last two performance groups, we analysed the testing traces and identified three primary reasons why the LLMs may have failed to perform the actions as expected:

- their context lengths are sometimes too short (the number of available tokens for LLM input and output) to consider all the page content;
- the LLM ability to extract page content is variable. Our code and prompts for performing this extraction do not take into account the structured representation of the data, such as tabular forms, on which assertion evaluation relies. Furthermore, our prompts could be more suited to some LLMs than others;
- Some NL test steps may be interpreted as ambiguous by certain LLMs and not by others. For instance, the assertion "Assert that the term 'Course' is present 4 times" is interpreted differently: some LLMs expect exactly four occurrences, while others interpret it as at least four. Additionally, the same LLM may have different interpretations of the same assertion across several runs.

Initial conclusions: 1) Considering these results together with Table 4, we can conclude that Llama 3.3 70B, on the 8 LLMs considered, demonstrates good capabilities in NL test case execution while maintaining high consistency (standard deviation < 0.15). 2) Evaluating LLM capabilities in performing navigation actions would benefit from more complex datasets. We intend to propose a dataset of NL test cases, in which each navigation step is associated with an expected GUI screenshot to yield more precise measurements of standard deviation. 3) The present study is subject to technical limitations, e.g., limited LLM context size or data extraction challenges, which may affect our conclusions. 4) NL test cases may include ambiguous steps. Our approach requires an additional step to identify and manage such ambiguities. 5) None of the LLMs used in our experiments are specialised for test case execution. LLMs specifically trained or fine-tuned for GUI interaction could provide better results.

6.2 RQ2: Predicted vs Observed Consistency

Setup: In RQ1, we identified three categories of LLMs based on their mean accuracies and selected three representative models: Llama 3 70B, as the most capable in executing NL test cases; Mistral Nemo 12B, as the least capable; and Qwen3 14B, which showed mixed performance.

We employ NLTestRunner, which implements Algorithm 1, to execute each test case and compute the estimated consistency measure, *consistency*. Each test case was executed in batches of 20 runs, from which an empirical consistency measure is derived based on the distribution of observed test verdicts. The tool was evaluated using the TestG suite, as well as two additional suites, TestM and TestO, developed to examine its applicability across two other open-source e-commerce platforms.

To compare the observed execution consistency of NL test cases with our measure *consistency*, we calculate the mean relative error (MRE) for each test case. MRE is a standard statistical measure used to quantify how far predictions deviate from reference values. Here, MRE quantifies how far the estimated consistency is from the observed one.

Results: Figure 6 presents, for each LLM, 3 box plots that show the distributions of MRE computed after the execution of each test case, for each test suite. The mean MRE values are depicted with crosses.

Aggregating across all test suites and LLMs, we obtain a mean MRE of 11%. Figure 6 shows a large difference in MRE between the models. Mistral Nemo 12B shows the highest mean MRE of 30%, Llama 3.3 70B and Qwen 3 14B achieve much lower values, with a mean MRE of 2%.

For Mistral Nemo 12B, *consistency*(tc) exhibits substantial deviations from the observed execution consistencies. Specifically, in one-third of the cases where the LLM agent produces the expected results, and in half of the cases where it does not, *consistency*(tc) underestimates the observed consistency. This indicates that when the ability of the LLM agent is limited, our measure *consistency*(tc) becomes inaccurate. In contrast, for LLM agents with mixed or strong capabilities in executing NL test cases, *consistency*(tc) produces estimates that closely align with the observed consistencies.

It is worth noting that, in the case of Qwen 3 14B, a high consistency score does not necessarily imply correct behaviour. Rather, it indicates that the model tends to behave very predictably, either producing the expected outcome very frequently, or failing to do so with the same consistency.

Initial conclusions: 1) Our experiments indicate that *consistency*(tc) is an accurate measure for estimating the execution consistency of a test case when the LLM has mixed or good abilities in NL test case executions. 2) For other LLMs, however, a more fine-grained metric, potentially combining multiple measures for different types of assertions or GUI interactions, could provide estimates that more closely align with the observed execution consistency.

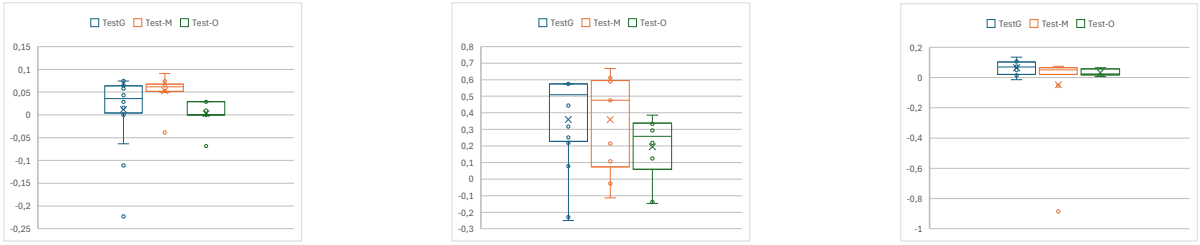
6.3 Threats To Validity

Internal threats. 1) The evaluation of LLM agents depends on our tools, which could be improved to better support response timeouts, capture structured data from GUIs, and verify the correctness of navigation actions. 2) The NL test suites we developed are made up of common interactions but do not cover all cases, such as hovering over or dragging elements. 3) While we took care to avoid writing ambiguous NL test steps, ambiguity is a difficult property to judge. Some works considered this problem [Dalpiaz et al., 2019], but further work is needed. 4) The design of the prompts used in our experiments may also limit the validity of our conclusions, particularly for other LLMs. Despite following a strict protocol and applying prompts consistently across models, they may still be better suited to some LLMs than to others.

External threats. 1) The *AUT*s we considered cover only 5 different web sites. Furthermore, while we included some negative assertions in the test suites, the *AUT*s themselves are not supposed faulty. To strengthen external validity, other application types (e.g. mobile applications) should be considered. 2) We considered 8 LLMs that can be deployed on local servers, but excluded cloud-based LLMs that may perform better for GUI interaction. We focused in this paper on locally deployable LLMs as they ensure confidentiality. These LLMs come from different companies, with model sizes ranging from 3B to 70B parameters. Future work could explore the evaluation of our tools with further LLMs. 3) We used the framework Stagehand for performing navigation actions, which is still evolving. New versions of Stagehand may improve results, especially for small LLMs (e.g., 7B). Other frameworks could also be considered. For example, we conducted some experiments with BrowserUse, a framework rather dedicated to the execution of general scenarios, but our initial results were not promising.

7 CONCLUSION

We investigated the feasibility of executing NL test cases for GUI applications using LLM agents. We introduced the notion of weak unsoundness and defined a measure to quantify the consistency of NL test case execution. Building on this, we developed an NL test case execution algorithm with guardrail mechanisms and specialised agents. Experiments on 8 LLMs of varying sizes indicate that current models can support GUI testing under specific conditions: unambiguous actions and assertions, and LLM agents able to perform navigation and evaluation tasks with high accuracy.



(a) Qwen 3 14B (b) Mistral Nemo 12B (c) Llama 3.3 70B

Figure 6: Box plots depicting the MRE measured used to quantify how far estimated consistencies deviate from real consistencies with the test suites with TestG, Test-M and Test-O.

In future research, our algorithms can be extended to generate NL test cases from scenarios, optimise agent compositions dynamically to GUI complexity, and include screenshots to better detect navigation issues.

REFERENCES

[Allamanis et al., 2018] Allamanis, M., Barr, E. T., Devanbu, P., and Sutton, C. (2018). A survey of machine learning for big code and naturalness. *ACM Comput. Surv.*, 51(4).

[Augusto et al., 2024] Augusto, C., Morán, J., Bertolino, A., de la Riva, C., and Tuya, J. (2024). Software system testing assisted by large language models: An exploratory study. In *Testing Software and Systems: 36th IFIP WG 6.1 International Conference, ICTSS 2024, London, UK, October 30 – November 1, 2024, Proceedings*, page 239–255, Berlin, Heidelberg. Springer-Verlag.

[Chen et al., 2024] Chen, Y., Hu, Z., Zhi, C., Han, J., Deng, S., and Yin, J. (2024). Chatunitest: A framework for llm-based test generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, FSE 2024, page 572–576, New York, NY, USA. Association for Computing Machinery.

[Dalpiaz et al., 2019] Dalpiaz, F., van der Schalk, I., Brinkkemper, S., Aydemir, F. B., and Lucassen, G. (2019). Detecting terminological ambiguity in user stories: Tool and experimentation. *Information and Software Technology*, 110:3–16.

[Deng et al., 2023] Deng, Y., Xia, C. S., Peng, H., Yang, C., and Zhang, L. (2023). Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, page 423–435, New York, NY, USA. Association for Computing Machinery.

[Godfrey, 2014] Godfrey, A. B. (2014). *Six Sigma*. John Wiley & Sons, Ltd.

[Hou et al., 2024] Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J., and Wang, H. (2024). Large language models for software engineering: A systematic literature review. *ACM Trans. Softw. Eng. Methodol.*, 33(8).

[Jard et al., 2000] Jard, C., Jéron, T., and Morel, P. (2000). *Verification of Test Suites*, pages 3–18. Springer US, Boston, MA.

[Liu et al., 2024] Liu, Z., Chen, C., Wang, J., Chen, M., Wu, B., Che, X., Wang, D., and Wang, Q. (2024). Make llm a testing expert: Bringing human-like interaction to mobile gui testing via functionality-aware decisions. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA. Association for Computing Machinery.

[Salva and Taguelmimt, 2025] Salva, S. and Taguelmimt, R. (2025). On the soundness and consistency of llm agents for executing test cases written in natural language. arXiv:2509.19136 [cs.SE]. https://arxiv.org/abs/2509.19136.

[Salva and Zafimiharisoa, 2014] Salva, S. and Zafimiharisoa, S. R. (2014). Model reverse-engineering of Mobile applications with exploration strategies. In *Ninth International Conference on Software Engineering Advances, ICSEA 2014*, Nice, France.

[Tretmans, 1996] Tretmans, J. (1996). Test generation with inputs, outputs and repetitive quiescence. *Software – Concepts and Tools*, 17(3):103–120.

[Wei et al., 2022] Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E. H., Le, Q. V., and Zhou, D. (2022). Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA. Curran Associates Inc.

[White et al., 2023] White, J., Fu, Q., Hays, S., Sandborn, M., Olea, C., Gilbert, H., Elnashar, A., Spencer-Smith, J., and Schmidt, D. C. (2023). A prompt pattern catalog to enhance prompt engineering with chatgpt. In *Proceedings of the 30th Conference on Pattern Languages of Programs*, PLoP '23, USA. The Hillside Group.

[Yoon et al., 2024] Yoon, J., Feldt, R., and Yoo, S. (2024). Intent-Driven Mobile GUI Testing with Autonomous Large Language Model Agents . In *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 129–139, Los Alamitos, CA, USA. IEEE Computer Society.