



Prompt engineering in LLMs for automated unit test generation: A large-scale study

Wendkûuni C. Ouédraogo¹ · Abdoul Kader Kaboré¹ · Yinghua Li²  · Haoye Tian⁴ · Anil Koyuncu⁵ · Jacques Klein¹ · David Lo³ · Tegawendé F. Bissyandé¹

Received: 31 March 2025 / Accepted: 3 March 2026

© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2026

Abstract

Unit testing is essential for software reliability, yet manual test creation is time-consuming and often neglected. Although search-based software testing improves efficiency, it produces tests with poor readability and maintainability. Although LLMs show promise for test generation, existing research lacks comprehensive evaluation across execution-driven assessment, reasoning-based prompting, and real-world testing scenarios. This study presents the first large-scale empirical evaluation of LLM-generated unit tests at the *full class level*, systematically analyzing four state-of-the-art models (GPT-3.5, GPT-4, Mistral 7B, and Mixtral 8x7B) against EvoSuite across 216,300 *generated* test cases targeting Defects4J, SF110, and CMD (a dataset mitigating LLM training data leakage). We evaluate five prompting techniques—Zero-Shot Learning (ZSL), Few-Shot Learning (FSL), Chain-of-Thought (CoT), Tree-of-Thought (ToT), and Guided Tree-of-Thought (GToT)—assessing syntactic correctness, compilability, hallucination-driven failures, readability, code coverage metrics, and test smells. Reasoning-based prompting particularly GToT significantly enhances test reliability, compilability, and structural adherence in general-purpose LLMs. However, hallucination-driven failures remain a persistent challenge, manifesting as non-existent symbol references, incorrect API calls, and fabricated dependencies, resulting in high compilation failure rates (up to 86%). Moreover, test smell analysis reveals that while LLM-generated tests are generally more readable than those produced by traditional tools, they still suffer from recurring design issues such as Magic Number Tests and Assertion Roulette, which hinder maintainability. Overall, our findings indicate that LLMs can serve as effective assistive tools for generating readable and maintainable test suites, but hybrid approaches that combine LLM-based generation with automated validation and search-based refinement are required to achieve reliable and production-ready test generation.

Keywords Automatic Test Generation · Unit Tests · Large Language Models · Prompt Engineering · Empirical Evaluation

Communicated by: Shin Yoo.

Extended author information available on the last page of the article

1 Introduction

Unit testing is fundamental to software quality, yet inadequate testing remains pervasive—with only 48% of developers regularly writing tests (Daka and Fraser 2014). While testing ensures correctness by validating components before integration (Beck 2000), the labor-intensive nature of test creation leads to significant coverage gaps in practice. Furthermore, when tests do exist, their poor structure often creates brittle, incomprehensible test suites that hamper maintenance (Almasi et al. 2017; Grano et al. 2018; Panichella et al. 2020).

Traditional automated test generation approaches have attempted to address these challenges, but each faces significant limitations. Search-Based Software Testing (SBST), particularly tools like EvoSuite (Fraser and Arcuri 2011), optimizes for code coverage but produces tests that are difficult to read (Almasi et al. 2017), contain unintuitive naming (Daka et al. 2015), and rely on generic assertions (Panichella et al. 2020). Most critically, these high-coverage tests often fail to effectively detect faults (Pinto and Vergilio 2010)—the fundamental purpose of testing. Alternative approaches like fuzzing (Zeller et al. 2019), random testing (Pacheco and Ernst 2007), and symbolic execution (Sen et al. 2005) each introduce their own trade-offs between coverage, precision, and scalability.

Large Language Models (LLMs) (Brown et al. 2020; Fan et al. 2023) have emerged as a promising alternative for test generation, potentially addressing traditional approaches' fundamental limitations. Initial studies suggest LLM-generated tests feature more natural naming conventions, human-like assertions, and better readability than SBST alternatives (Siddiq et al. 2024a; Tang et al. 2024; Yuan et al. 2023). Moreover, LLMs can generate both functional and exploratory tests beyond mere structural coverage (Wang et al. 2024a), aligning with comprehensive testing practices. Yet, it remains unclear whether prompting strategies that succeed in general code generation also transfer effectively to test generation. While reasoning-based prompts (e.g., CoT, ToT) have shown strong results in functional code tasks (Li et al. 2025; Wang et al. 2024b), unit test generation involves additional challenges: importing the class under test (CUT), setting up fixtures, managing program state, and writing reliable oracles (Hossain and Dwyer 2024; Molina et al. 2025). These aspects introduce domain-specific constraints absent in standard code synthesis.

In this work, we examine whether prompting hierarchies generalize to test generation, where structure, dependencies, and correctness criteria differ from conventional code synthesis. Our class-level evaluation across diverse benchmarks uncovers key limitations that must be addressed before LLM-based testing can be reliably deployed:

- **Impact of Prompt Engineering on Test Reliability and Usability:** Prior work (Siddiq et al. 2024a; Tang et al. 2024) has focused on minimal prompting strategies, leaving structured reasoning approaches (CoT, ToT, GToT) underexplored. We evaluate their effectiveness in improving test structure, compilability, and consistency.
- **Structural Correctness & Extractability of LLM-Generated Tests:** Prior work has evaluated various aspects of LLM-generated tests, including compilability and coverage (Siddiq et al. 2024a; Tang et al. 2024; Yang et al. 2024, and in some cases, test smells (Siddiq et al. 2024a), providing valuable insights into the runtime viability and maintainability of generated tests. However, few studies assess whether LLM-generated outputs are structurally valid and directly usable as standard JUnit test classes. We build on this line of work by applying extractability metrics (MSR and CSR Macedo et al. 2024)

to measure whether generated files conform to JUnit expectations and can be seamlessly integrated into real-world testing pipelines.

- **Readability, Maintainability & Test Smells:** LLM tests are considered more readable than SBST outputs (Yuan et al. 2023; Tang et al. 2024), but readability doesn't ensure maintainability. Test smells reduce long-term usability (Panichella et al. 2020). We assess maintainability using readability models (Scalabrino et al. 2018; Sergeyyuk et al. 2024), complexity metrics, and automated test smell detection, comparing LLMs with EvoSuite.
- **Generalizability Across Diverse Codebases:** Prior studies on LLM-generated tests rely on limited benchmarks (Siddiq et al. 2024a; Tang et al. 2024; Yang et al. 2024), raising concerns about generalizability. We address this by evaluating three datasets: Defects4J, featuring real-world buggy software; SF110, containing Java programs with existing test cases; and CMD, a curated dataset excluding projects released before May 2023 to mitigate LLM training data leakage. **This paper:** We systematically evaluate LLM-generated test suites and prompt engineering impact. Building on prior comparisons (Siddiq et al. 2024a; Tang et al. 2024; Yang et al. 2024), surveys (Wang et al. 2024a), and AI risk studies (Sallou et al. 2023), we conduct a large-scale evaluation across four LLMs and five prompting techniques against EvoSuite. Our study focuses on understanding prompting strategies in LLM-based test generation. We contrast LLM outputs with EvoSuite (Fraser and Arcuri 2011), the state-of-the-art in search-based test generation (SBST) for Java, which has consistently outperformed random testing and symbolic execution (Shamshiri et al. 2015). In the SBFT Tool Competition (Jahangirova and Terragni 2023), EvoSuite consistently ranks first in line/branch coverage and mutation score, outperforming Randoop (Pacheco and Ernst 2007), Kex (Abdullin and Itsykson 2022), and UTBot¹, and remains the most widely adopted and validated SBST tool (Almasi et al. 2017; Ouédraogo et al. 2025; Shamshiri et al. 2015). By positioning LLMs against EvoSuite, we ensure a strong, credible baseline (the de facto reference for automated Java test generation), providing a fair reference point for assessing whether LLMs, with proper prompting, can reach or surpass state-of-the-art SBST methods. Unlike method-level approaches in prior work, our class-level evaluation aligns with real-world testing workflows.

We make the following key contributions:

● **Systematic Investigation of Prompt Engineering:** We expand on previous research (Tang et al. 2024) by testing five structured prompting techniques (ZSL, FSL, CoT, ToT, GToT) with general-purpose LLMs, moving beyond the code-specific models in Yang et al. (2024). Unlike prior method-level studies (Chen et al. 2024; Siddiq et al. 2024a; Yang et al. 2024; Yuan et al. 2024) our class-level evaluation provides a more realistic assessment by capturing inter-method dependencies. Our results challenge claims (Yang et al. 2024) that reasoning-based prompting has limited impact on test generation for code-specialized LLMs. Our experiments demonstrate GToT consistently enhances test quality—improving structure, reducing hallucinations, and increasing extractability and compilability.

¹<https://www.utbot.org/>

② **Systematic Structural Correctness & Extractability Analysis:** Our analysis reveals that reasoning-based prompting improves test extractability and real-world usability, addressing inconsistencies in LLM test suite generation.

④ **Systematic Differentiation Between Readability & Maintainability:** We analyze how prompt engineering impacts both readability and long-term maintainability, highlighting areas where LLMs produce cleaner or more problematic test structures.

⑤ **Systematic Multi-Dataset, Multi-Model Evaluation for Generalizability:** We benchmark four LLMs on Defects4J, SF110, and CMD, offering a broader dataset scope than prior studies. CMD mitigates data leakage by including projects released after May 2023.

⑥ **Publicly Available Benchmark & Replication Package:** We provide a benchmark of 216,300 tests across 690 Java classes, with prompt templates, evaluation scripts, and results accessible at: <https://anonymous.4open.science/r/LLM4TS-0F76/>

The remainder of this paper is structured as follows: Section 2 covers LLM-based test generation, Section 3 outlines our experimental setup, Section 4 presents evaluation results, Section 5 discusses threats to validity, limitations and future work, Section 6 reviews related work, Section 7 discusses key findings and implications, and Section 8 concludes the study.

2 Background

In this section, we introduce the core concepts underpinning our study. We outline key ideas from prompt engineering, describe metrics adapted or extended for evaluating generated tests (e.g., MSR and CSR), clarify the distinction between readability and understandability, present common test smells, and review static analysis tools used to assess code quality. These notions provide essential context for interpreting the influence of prompting strategies on LLM-based test generation.

2.1 Prompt Engineering

Prompt engineering creates effective prompts that guide LLMs in specific tasks (Amatriain 2024; Naveed et al. 2023; Sahoo et al. 2024; Siddiq et al. 2024b). This approach is valuable for complex problems where single-line reasoning is insufficient (Amatriain 2024; Kojima et al. 2022; Wei et al. 2022), allowing LLMs to consider multiple possibilities before reaching conclusions. The significance of prompt engineering is twofold: specific prompts yield precise responses (Si et al. 2022), and well-crafted instructions control output format, reducing post-processing needs (Macedo et al. 2024). Software engineering applications typically use basic approaches like Zero-shot (Kojima et al. 2022) and Few-shot learning (Brown et al. 2020), though recent research has expanded to include Chain-of-Thought (Wei et al. 2022), self-consistency (Wang et al. 2022), and Tree-of-Thoughts (Wang et al. 2022).

Zero-shot learning enables models to generalize without examples but struggles with complexity (Sahoo et al. 2024). **Few-shot learning** uses examples to improve performance on novel tasks (Brown et al. 2020). **Chain-of-Thought** (CoT) (Wei et al. 2022) outlines intermediate reasoning steps, while **Self-consistency** (Wang et al. 2022) uses diverse reasoning paths to select consistent answers. **Tree-of-Thoughts** (ToT) (Long 2023) builds

on CoT by exploring a reasoning “tree” with search algorithms. This study applies these techniques to adapt LLMs for unit test generation.

2.2 Match Success Rate (MSR) and Code Extraction Success Rate (CSR)

Match Success Rate (MSR) and Code Extraction Success Rate (CSR), introduced by Macedo et al. (2024), evaluate the reliability of LLM-generated code. MSR measures the proportion of outputs where predefined patterns successfully extract source code, ensuring structural adherence. CSR assesses the proportion of valid test code extracted, filtering out explanatory text or formatting inconsistencies. These metrics are crucial for usability assessment, as LLM outputs often mix code with commentary (Fig. 1).

In our setting, we adopt these metrics as a preliminary lens: rather than asking whether the code is syntactically valid or compilable, we first ask whether test code can be consistently detected and cleanly extracted from raw LLM outputs. This allows us to decouple *format and extractability* from later concerns of syntax, compilation, and execution.

Rationale LLM outputs often interleave code with natural language (e.g., explanations, caveats, reformatted snippets). This interleaving can bias downstream compilation- or execution-based metrics if extraction is not controlled. Following Macedo et al. (2024), we decouple *format/extractability* from *syntax/compilation* using two format-oriented metrics tailored to JUnit tests.

Definitions (Adapted to Test Generation) Let \mathcal{O} be the set of all LLM responses for a given setting (model \times prompt \times class). Each response $o \in \mathcal{O}$ is treated as raw text. We compute:

- **MSR (Match Success Rate).** Fraction of responses where a JUnit-oriented *detector* finds a code segment that *appears* to delimit a test suite. Formally,

$$MSR = \frac{|\{o \in \mathcal{O} \mid detect(o) = true\}|}{|\mathcal{O}|}. \tag{1}$$

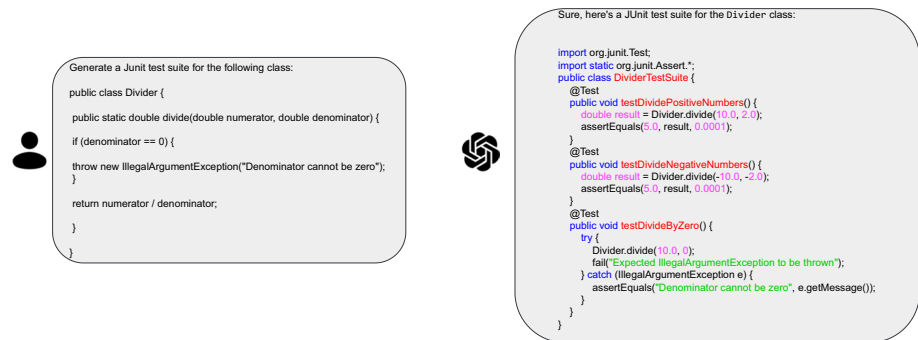


Fig. 1 A Sample Use of ChatGPT in Unit test Suite generation

Detection succeeds when either (i) explicit delimiters are respected (“###Test START###” ... “###Test END###”); or (ii) in their absence, fallback patterns match a plausible JUnit test file (package/imports/class declaration with @Test or JUnit imports). MSR is a *format-compliance* signal: it tells us whether a response contains something that looks like a test suite, not whether that suite is valid or compilable.

- **CSR (Code Extraction Success Rate).** Fraction of responses where extraction yields a *structured test file* that passes minimal *structural* sanity checks. Formally,

$$\text{CSR} = \frac{|\{o \in \mathcal{O} \mid \text{extract}(o) \rightarrow \hat{t} \wedge \text{is_structured}(\hat{t})\}|}{|\mathcal{O}|}. \quad (2)$$

The extraction function isolates the best candidate \hat{t} (single file) after stripping any non-code content. `is_structured(·)` verifies that the candidate looks like a *complete* JUnit test file (e.g., optional `package`, one or more `imports` including JUnit, a `public class` with at least one method or assertion). CSR is independent of syntax or compilation (both are assessed later in RQ2).

Detectors and Patterns We use a two-stage detector:

1. **Delimiter-first.** If the prompt requested explicit boundaries (“###Test START###” / “###Test END###”), we extract the enclosed block.
2. **Fallback patterns.** If delimiters are absent or malformed, we search for a *single* best candidate using ordered patterns that privilege JUnit-shaped files (package line, imports, class declaration, and at least one @Test or assertion).

If multiple code fences exist, we select the longest candidate that satisfies the JUnit shape. If no candidate passes detection, both MSR and CSR count that response as a failure.

Structural Validation for CSR `is_structured(\hat{t})` checks that the extracted block is not a partial snippet but a plausible file: (i) at least one `import` or a direct reference to JUnit/assertions; (ii) a single top-level `class` with balanced braces; (iii) absence of obvious non-Java artifacts (Markdown fences, HTML) within the extracted region.

Illustrative Example (Divider) Figure 1 illustrates a typical case: the user asks for a JUnit suite for a simple `Divider` class. ChatGPT returns (i) a valid JUnit test class `DividerTestSuite`, and (ii) a natural-language summary of the scenarios. In our framework, MSR=1 because the detector identifies a plausible test file, and CSR=1 because extraction isolates a structured class with imports, assertions, and balanced braces. The trailing explanation is ignored at this stage. This example shows why MSR/CSR are necessary: without controlled detection and extraction, downstream compilation (RQ2) would be contaminated by the extraneous prose.

What MSR/CSR do *not* Measure Neither metric judges Java syntax correctness, compilation, coverage, or fault detection. They are pre-execution format/extractability filters. As shown by Macedo et al. (2024), controlling output format can substantially affect downstream compilation; we therefore isolate this step as RQ1.

- Synthetic Corner Cases
1. **MSR = 1, CSR = 1.** Response has delimiters and encloses a full JUnit file.
 2. **MSR = 1, CSR = 0.** Detector matches a region (e.g., a Markdown fence), but extracted content fails structural checks.
 3. **MSR = 0, CSR = 0.** Response is purely explanatory or pseudo-code, no code region is detected.

Scope Note MSR/CSR are LLM-specific. SBST tools (e.g., EvoSuite) directly emit JUnit files by construction; applying MSR/CSR would always succeed and thus be uninformative. We therefore report MSR/CSR only for LLM outputs and use syntax/compilation checks for *both* paradigms in RQ2.

2.3 Readability vs. Understandability

Readability and understandability are distinct code quality aspects. Readability concerns visual clarity (indentation, naming, formatting), while understandability measures cognitive effort needed to comprehend logic (Buse and Weimer 2009; Grano et al. 2018; Oliveira et al. 2022; Winkler et al. 2024). Readability models evolved from traditional approaches (Buse and Weimer 2009; Daka et al. 2015; Dorn 2012; Posnett et al. 2011) to deep learning-based assessments (Mi et al. 2022). Though human evaluations are ideal, their scalability is limited by subjectivity and evaluator fatigue (Kumar et al. 2024; Sergeyuk et al. 2024; Sun et al. 2024; Zhang et al. 2024a).

Scalabrino et al. (2018) implemented a robust readability model integrating syntactic, structural, visual, and textual elements from Java projects. Using 104 features, it improved accuracy by 6.2% over previous methods and better matched human assessments Sergeyuk et al. (2024). Its training on JUnit projects makes it especially suitable for evaluating test code readability. Unit tests present unique challenges—poor readability obscures intent and hinders maintenance, particularly with complex fixtures (Grano et al. 2018; Winkler et al. 2024). Our study evaluates LLM-generated tests for readability, maintainability, and practical usability.

2.4 Test Smells

Unit testing verifies individual components' functionality while maintaining readability. Test smells indicate poor practices (Palomba et al. 2016; Peruma et al. 2019) that lead to fragility and tight coupling. Assertion Roulette and Magic Number Tests—documented in TsDetect²—commonly appear in both EvoSuite and LLM-generated tests. In Listing 1, Assertion Roulette shows multiple `assertThat()` calls without clarifications, obscuring failure sources.

²<https://testsmells.org/pages/testsmellexamples.html>

```

1 @MediumTest
2 public void testCloneNonBareRepoFromLocalTestServer() throws Exception {
3     Clone cloneOp = new Clone(false, integrationGitServerURIFor("small-repo.early.git"), helper().newFolder());
4     Repository repo = executeAndWaitFor(cloneOp);
5     assertThat(repo, hasGitObject("ba1f63e4430bf267d112b1e8afc1d6294db0cce"));
6     File readmeFile = new File(repo.getWorkTree(), "README");
7     assertThat(readmeFile, exists());
8     assertThat(readmeFile, oLength(12));
9 }

```

Listing 1 Example of Assertion Roulette Test Smell

In Listing 2, Magic Number Test demonstrates unexplained values (15, 30) without context, compromising maintainability.

```

1 @Test
2 public void testGetLocalTimeAsCalendar() {
3     Calendar localTime = calc.getLocalTimeAsCalendar(BigDecimal.valueOf(15.5D), Calendar.getInstance());
4     assertEquals(15, localTime.get(Calendar.HOUR_OF_DAY));
5     assertEquals(30, localTime.get(Calendar.MINUTE));
6 }

```

Listing 2 Example of Magic Number Test Test Smell

We use TsDetect (Peruma et al. 2020) to analyze how prompt engineering affects test smell prevalence in LLM-generated tests compared to EvoSuite-generated tests.

2.5 Static Analysis Tools: Checkstyle, PMD, and SpotBugs

To assess the structural quality of LLM-generated test suites, we rely on three widely used static analysis tools: Checkstyle³, PMD⁴, and SpotBugs⁵

Checkstyle Checkstyle is a static code analysis tool that verifies Java code conformance to coding standards. It detects formatting issues, naming convention violations, and structural inconsistencies. In our study, we adopt both the *Sun Code Conventions*⁶ and *Google Java Style*⁷ profiles to evaluate coding standard adherence.

PMD PMD is a source code analyzer that detects common code issues such as unused variables, empty catch blocks, and overly complex code. It provides metrics for both **cyclomatic complexity** (control flow paths) and **cognitive complexity** (human-perceived difficulty). Following SonarSource guidelines⁸, we classify *cognitive complexity* into: *Low* (< 5), *Moderate* (5–10), *High* (11–20), and *Very High* (≥ 21). For *cyclomatic complexity* (based on PMD thresholds), we use: *Low* (1–4), *Moderate* (5–7), *High* (8–10), and *Very High* (≥ 11). These thresholds allow us to assess whether generated test cases remain structurally simple and maintainable.

³ <https://checkstyle.sourceforge.io>

⁴ <https://pmd.github.io/>

⁵ <https://spotbugs.github.io/>

⁶ <https://www.oracle.com/java/technologies/javase/codeconventions-contents.html>

⁷ <https://google.github.io/styleguide/javaguide.html>

⁸ <https://www.sonarsource.com/docs/CognitiveComplexity.pdf>

SpotBugs SpotBugs detects potential software bugs through bytecode-level analysis. It categorizes issues based on severity into four ranked priority levels:

- **Scariest:** Critical defects that threaten correctness, security, or stability.
- **Scary:** Severe bugs that may cause subtle but impactful failures.
- **Troubling:** Moderate issues, potentially leading to runtime problems if left unaddressed.
- **Of Concern:** Minor issues or code smells that may reduce code quality but rarely break functionality.

This categorization allows us to distinguish not only the number of detected issues, but also their potential severity and impact on code reliability. By using these severity levels, we can better characterize the trade-offs between code structure and safety in automatically generated test suites.

3 Experimental Setup

3.1 Overview

Our methodology (Fig. 2) evaluates LLM-generated test suites using ZSL, FSL, CoT, ToT, and GToT prompting techniques. To assess prompt engineering and reliability (RQ1, RQ2), we analyze structural correctness and usability using MSR, CSR, syntax correctness, and compilability. For readability and maintainability (RQ3, RQ4), we employ automated metrics including a deep learning model (Scalabrino et al. 2018) that correlates with human assessments (Sergeyuk et al. 2024), alongside complexity metrics and coding standard adherence. For code coverage (RQ5), we compare LLM tests with EvoSuite across line, instruction, and method coverage. Finally, we evaluate test smell prevalence (RQ7) to determine if reasoning-based prompting improves maintainability.

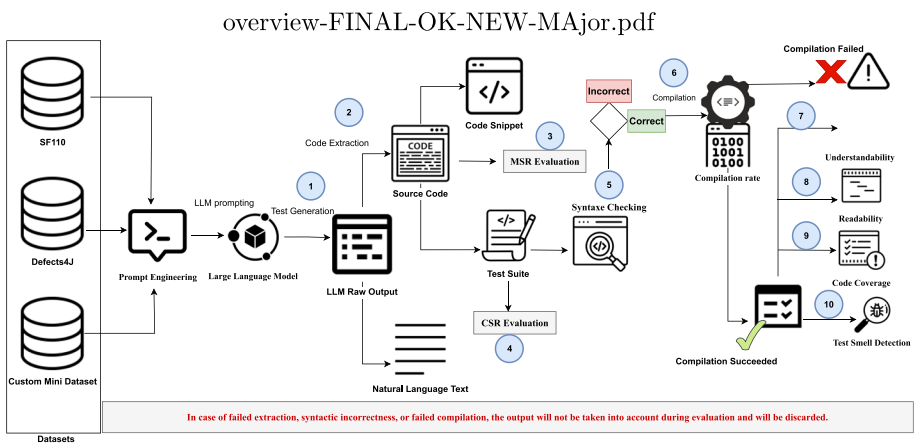


Fig. 2 Overview of the pipeline used to design the experiments

3.2 Research Questions

This study evaluates LLMs in test generation and the impact of prompt engineering through seven research questions:

RQ0: What is the ability of SBST (EvoSuite) and LLMs to generate test code at all? Before analyzing correctness, readability, or maintainability, it is essential to first assess whether different paradigms can actually produce test code for a given dataset. We therefore examine the raw generation capacity of EvoSuite and LLMs across datasets, measuring how many test suites were produced out of the total expected runs. This provides a baseline for understanding the feasibility of test generation before deeper analyses.

RQ1: How do prompt engineering techniques impact the output format and extractability of LLM-generated test suites? We examine how different prompting strategies influence test suite structure and extractability through ❶ Match Success Rate (MSR), measuring detection of test code via predefined patterns, and ❷ Code Extraction Success Rate (CSR), evaluating reliable extraction of valid test code. These metrics determine adherence to JUnit structures and isolation from extraneous content.

RQ2: How does prompt engineering influence the generation of syntactically correct and compilable test suites by LLMs, in comparison with SBST (EvoSuite)? We assess how different prompting strategies affect the ability of LLMs to generate test suites that ❶ adhere to syntax rules and ❷ successfully compile, enabling execution without ensuring functionality. In addition, we contrast these results with EvoSuite as a representative SBST baseline, in order to highlight similarities and differences between search-based and LLM-based test generation with respect to syntactic validity and compilability.

RQ3: Beyond syntactic correctness, how well do LLMs generate human-understandable and maintainable test suites? We evaluate if LLM-generated tests: ❶ convey test logic clearly; ❷ remain adaptable over time; and ❸ integrate seamlessly into workflows, determining if they meet practical development standards. EvoSuite is not included in RQ3 analyses, as its primary goal is maximizing structural coverage rather than producing understandable or maintainable tests. This distinction has been repeatedly emphasized in the literature: Rojas et al. (2015) show that while EvoSuite increases coverage and reduces testing time, developers consistently struggle with the readability of its outputs. Similarly, Shamshiri et al. (2018) report that although developers are as effective with EvoSuite-generated tests as with manual ones, they take significantly longer to maintain them and perceive them as harder to understand. These findings confirm that readability and understandability are not baselines EvoSuite provides, but rather limitations that have motivated recent efforts to refactor generated tests with LLMs (Biagiola et al. 2025; Deljouyi et al. 2025). Therefore, RQ3 focuses exclusively on LLM-generated tests, where prompt design explicitly targets human-oriented quality dimensions such as readability and maintainability.

RQ4: How do LLM-generated test suites compare to EvoSuite and align with developer expectations in terms of readability? We investigate whether LLM tests meet developer expectations by: ❶ enhancing navigation through clear naming and structure; ❷ reducing ambiguity with well-organized presentation; and ❸ supporting collaboration. We analyze how prompting techniques influence readability using an automated model correlated with human evaluation. While EvoSuite was excluded in RQ3 because static analysis tools such as Checkstyle, PMD, and SpotBugs assess dimensions (style compliance, cyclomatic/

Table 1 Summary of LLMs Used in Current Study

Model Name	# Parameters (Billions)	Context Length (Tokens)	Release Date
GPT 3.5-turbo (OpenAI 2023)	175	4 096	Nov, 2022
GPT 4 (Achiam et al. 2023)	>175	8 192	Mar, 2023
Mistral 7B (Jiang et al. 2023)	7	4 096	Sep, 2023
Mixtral 8x7B (Jiang et al. 2024)	46.7	8 192	Dec, 2023

cognitive complexity, and bug patterns) that it was never designed to optimize, making a direct comparison unfair and potentially misleading, in RQ4 we include EvoSuite as a baseline reference. Here, the focus is on readability from a developer perspective, evaluated with an established readability model (Scalabrino et al. 2018). EvoSuite's widespread adoption in SBST makes it a useful empirical point of comparison to determine whether LLM-generated tests deliver tangible gains in human-oriented quality, complementing the structural strengths of SBST with improved clarity and readability.

RQ5: How Thorough Are LLM-Generated Test Suites in Terms of Code Coverage?

We examine code coverage (line, instruction, method) of LLM-generated tests compared to EvoSuite (Fraser and Arcuri 2011), analyzing how prompting strategies impact exploration of program behavior.

RQ6: How do LLM-generated test suites compare to EvoSuite-generated test suites in terms of test smell prevalence? We compare the absence of test smells in LLM-generated tests against EvoSuite to determine whether prompt engineering produces more maintainable tests with fewer design issues than traditional SBST techniques.

3.3 Large Language Models

Our study focuses on four instruction-tuned decoder-only models known for their enhanced ability to follow instructions compared to base models. These models belong to the OpenAI and MistralAI families and are detailed in Table 1.

GPT-3.5-turbo has a token limit of 4,096, and exceeding this limit can hinder class context understanding, affecting test quality (Tang et al. 2024). To ensure fairness, all models adhere to this limit, excluding inputs that exceed it. Token counts are computed using OpenAI's tiktoken⁹ library.

We select OpenAI models (GPT-3.5, GPT-4) and MistralAI models (Mistral 7B, Mixtral 8x7B) for their relevance and efficiency. OpenAI models ensure comparability with prior LLM-based test generation studies (Siddiq et al. 2024a; Tang et al. 2024; Yang et al. 2024). MistralAI models, underexplored in this domain, offer cost-effective alternatives: Mistral 7B matches Llama 34B's performance (Jiang et al. 2023), while Mixtral 8x7B, leveraging Sparse Mixture-of-Experts (SMoE), surpasses Llama 2 70B with lower inference costs (Jiang et al. 2024). This selection balances established and emerging models, broadening the scope of LLM-based test generation research.

⁹<https://github.com/openai/tiktoken>

Furthermore, all models are set to a standard temperature of 0.7. Our selection criteria prioritized models that excel in code generation and possess accessible APIs, making them suitable for both academic research and industrial applications (Fan et al. 2023; Wang et al. 2024a).

3.4 Prompt Design

Unit test generation research emphasizes practical and structured approaches (Chen et al. 2023; Siddiq et al. 2024a; Tang et al. 2024; Yuan et al. 2023). To ensure clarity, we design prompts with varying complexity using a standardized template. Each prompt comprises: (1) a Natural Language Description (NLD) specifying the task, and (2) a Source Code placeholder for inserting the Class Under Test (CUT). The following sections detail these components and their alignment with unit testing practices.

- **Natural Language Description:** We design Natural Language Descriptions (NLDs) that guide LLMs in test generation, including: (i) a role-playing directive simulating professional testers (Chen et al. 2023; Tang et al. 2024; ii) a task-specific directive for JUnit 4 test file generation; and (iii) a control statement enforcing structured output with markers. We introduce delimiters (###Test START### and ###Test END###) to separate test code from explanatory text.
- **Source Code:** Aligned with prior work on learning-based unit test generation (Chen et al. 2023; Tang et al. 2024), our prompts vary by technique, either including (i) only the class name and source code or (ii) an additional example class with its test suite to provide context and comparative guidance.

This study evaluates five prompt techniques: Zero-Shot Learning (ZSL), Few-Shot Learning (FSL), Chain-of-Thought (CoT), Tree-of-Thought (ToT), and Guided Tree-of-Thought (GToT). ZSL uses minimal context for generalization, while FSL introduces examples to improve structure. For consistency, we use the same medium-sized class across experiments, ensuring token limit compliance for FSL examples. CoT enhances reasoning through step-by-step prompting, while ToT explores multiple reasoning paths before selecting test cases. GToT, our novel approach, simulates collaborative reasoning among three “testers” who iteratively refine tests, combining CoT’s structured reasoning with ToT’s diversity to improve coverage, correctness, and variability. Figure 3 illustrates CoT and ToT prompts, while Fig. 4 shows GToT’s collaborative structure. All prompts are available in our replication package.

3.5 The EvoSuite SBST tool

For the SBST baseline, we selected EvoSuite (Fraser and Arcuri 2011) as it is the most mature and empirically validated tool for automated test generation in Java. Recent editions of the SBFT competition confirm EvoSuite’s leadership over competitors such as Randoop (Pacheco and Ernst 2007), Kex (Abdullin and Itsykson 2022), and UTBot, particularly in terms of branch and line coverage (Jahangirova and Terragni 2023). Large-scale empirical analyses (e.g. Shamshiri et al. 2015) also show that EvoSuite can detect real faults in Defects4J more effectively than random or commercial tools. Moreover,

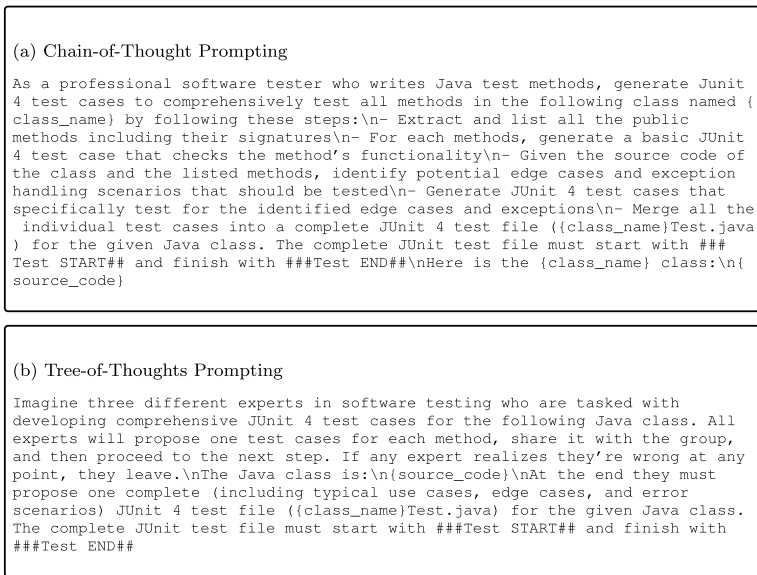


Fig. 3 Examples of Chain-of-Thought and Tree-of-Thoughts for test suite generation

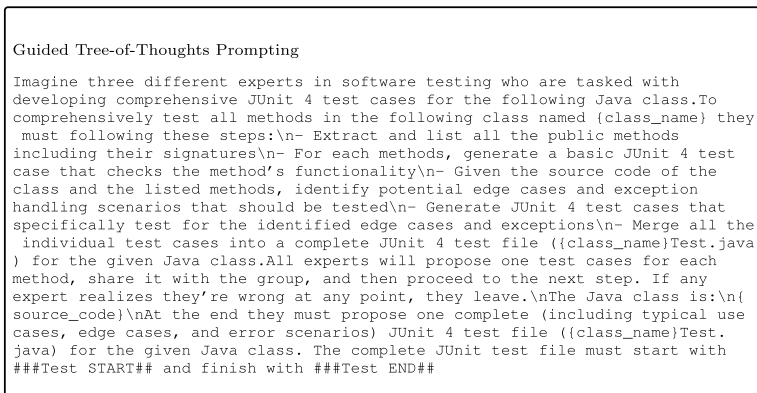


Fig. 4 Example of Guided Tree-of-Thoughts (GTOT) for test suite generation

recent research on enhancing automated test generation (Ouédraogo et al. 2025) continues to adopt EvoSuite as the reference point when proposing extensions or hybrid approaches. These consistent results across competitions, benchmarks, and empirical studies justify the choice of EvoSuite as the most reliable SBST comparator for our study. We employ its default settings, including the DynaMOSA algorithm (Panichella et al. 2017), which optimizes branch, method, and line coverage. To mitigate the randomness of genetic algorithms, we run EvoSuite 30 times per class, following (Tang et al. 2024). A three-minute time budget is allocated per test generation, aligning with prior studies (Arcuri and Fraser 2013; Fraser et al. 2015; Shamshiri et al. 2015), which find this duration sufficient

for effective test case generation. We deliberately disable EvoSuite's post-generation processing, which reduces test smells, to ensure a fair comparison with unoptimized LLM-generated tests. This approach allows for an unbiased evaluation of the baseline capabilities of both methods, highlighting their inherent strengths and limitations.

Unit of Analysis Each EvoSuite run produces exactly one test suite, which corresponds to a single Java test file. Therefore, all counts and percentages reported in our study are based on the number of generated test files (one per suite). This ensures consistency with our treatment of LLM-generated outputs, where each model-prompt-iteration also produces one test suite (file).

Hardware For reference, EvoSuite experiments are conducted on an AMD EPYC 7552 48-Core Processor running at 3.2 GHz with 640 GB of RAM. However, it's important to note that not all available resources are necessarily utilized during the tests.

3.6 Datasets

To minimize selection bias and ensure generalizability, we employ three diverse datasets: SF110, Defects4J, and a Custom Mini Dataset (CMD). Both SF110 and Defects4J are used to evaluate both EvoSuite and LLM performance. The CMD, containing GitHub projects created after May 2023, is exclusively tested with LLMs due to compatibility issues with EvoSuite's Java version requirements¹⁰. This dataset focuses on well-maintained projects from the OceanBase Developer Center (ODC)¹¹ and Conductor OSS¹².

While SF110 covers academic and industrial Java programs and Defects4J focuses on real-world bug-fixing scenarios, both datasets were filtered with token constraints so that every retained class fits all prompt engineering techniques (ZSL, FSL, CoT, ToT, GToT). This reduced SF110 from 346 to 182 classes (74 projects) and Defects4J to 16 projects (477 classes). We built CMD to complement SF110 (academic/industrial programs) and Defects4J (bug-fixing scenarios) with contemporary, evolving software. It contains two actively maintained projects ODC (OceanBase Developer Center) and Conductor OSS, selected to assess generalization beyond legacy-style codebases. CMD is carefully curated under three constraints: (i) token limits so each class fits all prompt engineering techniques (ZSL, FSL, CoT, ToT, GToT), (ii) only classes committed after May 2023 to reflect modern Java practices and reduce potential training leakage, and (iii) a buggy-fixed pairing scheme (as in Defects4J) to enable controlled evaluation. ODC (13 releases, 527 stars, 20 contributors) is a database development tool providing SQL development, risk management, and data security; Conductor OSS (11 releases, 15.5k stars, 53 contributors) is a microservices orchestration framework originally developed at Netflix. Despite its smaller scale, CMD offers a modern, actively maintained testbed that complements SF110 and Defects4J.

¹⁰<https://github.com/EvoSuite/evosuite/issues/433>

¹¹<https://github.com/oceanbase/odc>

¹²<https://github.com/conductor-oss/conductor>

Table 2 Overview of datasets and their complexity

Dataset	Description	Number of Projects	Number of Classes	Complexity Overview		
				Low	Moderate	High
SF110	Derived from DynaMOSA benchmark (Panichella et al. 2017), originally 117 projects (346 classes). After applying token constraints to ensure all prompts fit within context windows, 74 projects (182 classes) were retained.	74	182	170	138	120
Defects4J	Contains 835 reproducible bugs across 17 open-source projects. Similarly, token constraints were applied to guarantee that each selected class can accommodate all prompt engineering techniques without exceeding context limits, resulting in 16 projects (477 classes).	16	477	254	177	130
Custom mini Dataset (CMD)	Includes ODC, a collaborative project to enhance database functionalities, and Conductor OSS, an open-source project management tool aimed at efficiency and scalability. To ensure fairness and prevent potential training leakage, CMD was curated under three constraints: (i) token limits, so that each class can be used across all prompt engineering techniques (ZSL, FSL, CoT, ToT, GToT), (ii) only classes committed after May 2023, and (iii) a buggy-fixed pairing approach similar to Defects4J.	2	31	16	9	5

Notice. Reported classes correspond to those retained after applying filtering constraints (token limits and, for CMD, post-2023 commits and buggy-fixed pairing). Due to budgetary reasons, we only run GPT-4 and Mixtral-8x7B on the smallest dataset (CMD). Other models (GPT-3.5-Turbo and Mistral-7B) are run on all datasets

The datasets encompass projects with code of varying levels of cyclomatic complexity (detailed in Table 2), ranging from low to high. This diversity helps mitigate potential overfitting on specific complexity profiles and provides valuable insights into the potential of LLMs for real-world automated test case generation.

4 Experimental Results

4.1 [RQ0]: Ability of EvoSuite and LLMs to generate test code

[Experiment Goal]: This research question evaluates the fundamental ability of EvoSuite and LLMs to generate test suites, independently of syntactic correctness or compilability. The goal is to establish whether each paradigm can produce test code at scale for the target datasets, thereby clarifying their baseline feasibility as automated test generators.

[Experiment Design]: For each dataset and each class under test, we performed 30 independent generations. EvoSuite was applied on Defects4J and SF110 but not on CMD due to version compatibility issues with Java (Section 3.6). For LLMs, GPT-3.5-turbo and Mistral 7B were evaluated across all datasets, while GPT-4 and Mixtral 8×7B were restricted to CMD due to budget constraints (Section 3.6). The total number of expected test files was computed as the number of classes multiplied by 30 runs. For LLMs, since five prompting strategies were applied, the expected count was further multiplied by 5. Each generation, whether by EvoSuite or by an LLM, produces exactly one test suite, which corresponds to a single Java test file. Therefore, all counts and ratios in RQ0 are calculated on the basis of test files. The outcome metric is the ratio of generated test suites (i.e., outputs containing code) over the expected total, providing a direct measure of raw generation ability.

[Experiment Results]:

Table 3 summarizes the raw generation capacity of EvoSuite and the LLMs across datasets. On Defects4J (477 classes), EvoSuite produced tests in 77.6% of the expected runs (11,100/14,310). Notably, for two entire projects (Closure and Mockito), EvoSuite did not generate any tests for any class despite multiple attempts. In contrast, the LLMs generated code more frequently in this raw sense: GPT-3.5-turbo at 84.7% and Mistral 7B at 91.1% of the expected outputs. On SF110 (182 classes), EvoSuite achieved 100% generation (5,460/5,460), while GPT-3.5-turbo and Mistral 7B reached 84.4% and 94.0%, respectively. On CMD (31 classes), only LLMs were evaluated due to Java version incompatibilities with EvoSuite; GPT-3.5-turbo approached saturation at 99.6% (4,633/4,650), GPT-4 and Mistral 7B were slightly lower (93.7% and 90.9%), and Mixtral 8×7B lagged at 63.4%.

RQ0 provides a necessary preliminary step by clarifying whether different paradigms can even produce test code before deeper quality analyses. The results highlight two important observations. First, EvoSuite’s generation capacity is not uniform across datasets: while it attains perfect generation on SF110, it fails to produce tests for certain Defects4J classes, supporting concerns raised in prior work that SBST tools may not always yield outputs for all classes under test (Arcuri et al. 2017; Fraser and Arcuri 2013; Ouédraogo et al. 2025). Second, LLMs show high raw generation rates across datasets, often approaching or surpassing EvoSuite in this preliminary dimension. However, generation rates differ markedly across models, as illustrated by Mixtral 8×7B’s weaker performance on CMD, indicating that raw generation ability is model- and context-sensitive.

These results establish RQ0 as a necessary baseline. Even mature SBST tools like EvoSuite do not always produce tests in challenging settings such as parts of Defects4J, which has direct consequences for downstream quality and coverage analyses. At the

Table 3 Test generation capacity of EvoSuite and LLMs across datasets

Dataset	EvoSuite	GPT-3.5	GPT-4	Mistral	Mixtral
Defects4J	11100 /	60565 /	–	65180 /	–
	14310	71550		71550	
	(77.6%)	(84.7%)		(91.1%)	
SF110	5460 / 5460	23048 /	–	25664 /	–
		27300		27300	
	(100%)	(84.4%)		(94.0%)	
CMD	–	4633 /	4356 /	4226 /	2947 /
		4650	4650	4650	4650
		(99.6%)	(93.7%)	(90.9%)	(63.4%)

* Percentages refer to successful generations

*Mixtral = Mixtral 8×7B;
Mistral = Mistral 7B; GPT-3.5 =
GPT-3.5-turbo

same time, the consistently high generation rates achieved by LLMs highlight the need to scrutinize not only whether code is produced, but also *what* is produced in terms of structure, syntactic correctness, compilability, readability, smells, and fault-detection capability before assessing practical viability. Finally, the divergence across models and datasets—for instance, the comparatively weaker performance of Mixtral 8×7B on CMD—demonstrates that raw generation capacity is strongly model- and context-sensitive, underscoring the importance of analyzing prompting effects (RQ1) and subsequent validity dimensions (RQ2–RQ6) while carefully distinguishing performance across different models and datasets.

Finding 1: EvoSuite, despite being a mature SBST tool, does not always generate tests for all classes (e.g., failures on `Closure` and `Mockito` in Defects4J), achieving 77.6% of expected outputs, while it reaches 100% on SF110. In contrast, LLMs display high raw generation capacity across datasets (often above 90%), though with variability across models (e.g., Mixtral 8×7B at 63.4% on CMD).

Relation to Prior Work Comparative studies with EvoSuite (e.g., Siddiq et al. (2024a); Tang et al. (2024); Yang et al. (2024)) typically report compilability or coverage after some code is returned, often on method-level targets or on filtered-class subsets (e.g., classes reduced to a single focal method with injected imports). In contrast, RQ0 isolates the raw question “does the tool emit an output file at all?” at the class level, where “per-class” means one generation per entire class under test, including all interdependent methods, with no import injection. Under this stricter setup across heterogeneous projects, we observe cases where EvoSuite produces no suite for some classes or projects, whereas LLMs usually emit some code output. This qualifies baseline feasibility and highlights the impact of granularity and dataset scope on raw generation.

Summary of RQ0: Even mature SBST tools like EvoSuite do not always return test suites for all classes, especially on heterogeneous datasets such as Defects4J. In contrast, LLMs almost always emit an output file, though at this stage we do not yet establish whether the content is valid test code. RQ0 therefore provides a baseline feasibility check: it clarifies whether tools emit outputs at all, before RQ1–RQ2 investigate format adherence, extractability, syntactic correctness, and compilability.

4.2 [RQ1]: Impact of prompt engineering on the output format and extractability.

[Experiment Goal]: This research examines how prompt engineering affects the format and extractability of LLM-generated test suites, evaluated through Match Success Rate (MSR) for detecting test code and Code Extraction Success Rate (CSR) for extracting structured test code, regardless of its syntactic correctness or compilability.

These metrics are specific to LLMs, since unlike LLMs, EvoSuite directly produces JUnit test files without extraneous natural language or prompt-driven variability. As a result, MSR and CSR are not applicable to SBST tools and are only reported for LLM-generated outputs.

[Experiment Design]: We evaluate prompt designs using Match Success Rate (MSR) and Code Extraction Success Rate (CSR). MSR measures adherence to predefined structures, while CSR assesses extraction of structured test code excluding commentary. To address LLMs’ tendency to mix code with explanations (Fig. 1), we introduced

delimiters (###Test START## and ###Test END##) to improve extraction accuracy (Fig. 3). Following Macedo et al. (2024), we used regular expressions for extraction. For CSR validity, test suites needed key components like import statements, ensuring completeness rather than isolated snippets. CSR evaluates extractability independent of syntax correctness or compilability. Experiments on SF110, Defects4J, and CMD assessed LLM performance across diverse scenarios.

[Experiment Results]:

Our study generated 216,300 files, with 190,619 (88.13%) containing source code.

Table 4 reports aggregated MSR and CSR scores by model and dataset, computed across all five prompting strategies (ZSL, FSL, CoT, ToT, GToT). This gives a global view of each model's ability to produce extractable test code on average. The detailed breakdown by prompting technique is presented in Tables 5 and 6, allowing fine-grained analysis of how each prompt affects format compliance and extractability.

Dataset-Level Performance. LLMs showed varying MSR and CSR scores across datasets (Table 4). GPT-3.5-turbo achieved the highest MSR (99.64%) and CSR (90.79%) on CMD, indicating strong structured output generation. GPT-4 also performed well, reaching a CSR of 83.59% on CMD. Mistral 7B excelled in SF110 (MSR: 94.00%) and Defects4J (MSR: 91.10%), demonstrating better structural adherence in these datasets. In contrast, Mixtral 8x7B struggled on CMD, with lower MSR and CSR scores.

Finding 2: OpenAI models (GPT-3.5, GPT-4) demonstrate superior MSR and CSR, indicating their ability to generate structured, extractable test suites. Mistral 7B also performs well, whereas Mixtral 8x7B struggles, highlighting model-specific variations.

Impact of Prompting Techniques. GToT consistently achieved the highest MSR across all models and datasets (Table 5), reinforcing its effectiveness in enforcing structural adherence.

For GPT-3.5-turbo, it reached **24.54%** on Defects4J, **23.95%** on SF110, and **20.01%** on CMD. Mistral 7B also benefited from GToT, achieving **20.79%** on Defects4J and **21.07%** on SF110. GPT-4 reached **18.20%** on CMD. ZSL showed competitive results in certain settings, with GPT-3.5-turbo reaching **21.51%** on Defects4J and **20.05%** on CMD. In contrast, FSL consistently produced lower MSR scores, suggesting that additional examples do not necessarily enhance structural adherence. ToT outperformed CoT across most models, improving MSR by up to 2.51% on Defects4J for GPT-3.5-turbo, highlighting the advantage of stepwise reasoning in structuring outputs.

Finding 3: GToT is the most effective prompting strategy for improving adherence to follow prompt instructions (MSR), while ZSL remains a strong baseline, especially for OpenAI models. FSL underperforms, challenging assumptions that more examples lead to better structural adherence.

GToT often achieved the highest CSR scores across datasets (Table 6), particularly for GPT-3.5-turbo and GPT-4, demonstrating its effectiveness in reducing extraneous content and structuring outputs. For example, GPT-3.5-turbo reached **24.54%** on Defects4J and **23.65%** on SF110 with GToT. However, CSR improvements are more sensitive to model-dataset interactions than MSR, and simpler prompting strategies can sometimes match or exceed GToT. On CMD, for instance, GPT-3.5-turbo achieved **21.70%** with ZSL compared to **21.06%** with GToT, and GPT-4 also reached over **23%** CSR with ZSL. For Mistral 7B,

Table 4 MSR and CSR by LLM and Dataset (aggregated over prompts)

Model Name	# of Generated Files				# of Files with code				MSR				CSR			
	SF110	Defects4J	CMD	SF110	Defects4J	CMD	SF110	Defects4J	CMD	SF110	Defects4J	CMD	SF110	Defects4J	CMD	
GPT 3.5-turbo	27 300	71 550	4 650	23 048	60 565	4 633	84.42%	84.65%	99.64%	78.10%	78.34%	90.79%				
GPT 4*	-	-	4 650	-	-	4 356	-	-	93.68%	-	-	83.59%				
Mistral 7B	27 300	71 550	4 650	25 664	65 180	4 226	94.00%	91.10%	90.88%	79.97%	80.35%	83.61%				
Mixtral 8x7B*	-	-	4 650	-	-	2 947	-	-	63.38%	-	-	53.57%				
Total/Average (%)	54 600	143 100	18 600	48 712	125 745	16 162	89.21	87.88	86.90	79.04	79.35	77.89				

*Note that due to budget constraints, we only paid to query GPT-4 and Mixtral 8x7B for CMD

Table 5 MSR by LLM and Prompt Engineering

Model Name	Defects4J (%)				SF110 (%)				CMD (%)						
	ZSL	FSL	CoT	ToT	GToT	ZSL	FSL	CoT	ToT	GToT	ZSL	FSL	CoT	ToT	GToT
GPT 3.5-turbo	21.51	16.43	17.51	20.02	24.54	19.95	17.97	19.40	18.73	23.95	20.05	20.05	19.94	19.94	20.01
GPT 4	–	–	–	–	–	–	–	–	–	–	21.35	21.30	21.03	18.11	18.20
Mistral 7B	19.72	19.31	20.07	20.10	20.79	19.72	19.39	19.63	20.19	21.07	20.04	19.88	20.04	19.83	20.21
Mixtral 8x7B	–	–	–	–	–	–	–	–	–	–	20.04	19.88	20.04	19.83	20.21

Table 6 CSR by LLM and Prompt Engineering

Model Name	Defects4J (%)						SF110 (%)						CMD (%)					
	ZSL	FSL	CoT	ToT	GToT		ZSL	FSL	CoT	ToT	GToT		ZSL	FSL	CoT	ToT	GToT	
GPT 3.5-turbo	21.51	16.43	17.51	20.02	24.54		21.30	17.30	18.05	19.70	23.65		21.70	20.77	20.09	16.39	21.06	
GPT 4	-	-	-	-	-		-	-	-	-	-		23.00	22.90	20.68	17.60	15.82	
Mistral 7B	20.26	20.23	20.65	20.13	18.73		21.25	21.15	20.95	20.43	16.23		20.60	20.29	20.37	20.11	18.62	
Mixtral 8x7B	-	-	-	-	-		-	-	-	-	-		18.51	23.36	18.95	15.58	23.60	

CoT slightly outperformed GToT on Defects4J (20.65% vs. 18.73%). FSL consistently underperformed across all settings, and Mixtral 8x7B remained the weakest overall despite peaking at **23.60%** CSR on CMD under GToT.

Finding 4: GToT often yields the highest CSR scores, but improvements vary depending on the model and dataset. ZSL remains competitive for GPT models sometimes even outperforming GToT while FSL consistently underperforms. Mixtral 8x7B remains the least effective overall, despite isolated gains under GToT.

To address LLMs' tendency to mix test code with explanatory text, we introduced delimiters (###Test START### and ###Test END###) to guide structured output. Despite these clear boundaries, no LLM consistently follows the expected format, with MSR varying between 16% and 25% across models and datasets (Table 6). This inconsistency directly impacts CSR, which ranges from 15% to 24% (Table 6), highlighting persistent challenges in reliably extracting complete and structured test code.

Finding 5: Explicit delimiters improve extraction by structuring LLM outputs, but all LLMs exhibit inconsistencies in following formatting instructions, with some failing more frequently, making automated extraction challenging.

Relation to Prior Work Existing studies on LLM-based test generation (e.g., Chen et al. (2024); Siddiq et al. (2024a); Tang et al. (2024); Yang et al. (2024); Yuan et al. (2024)) typically report results on syntactic correctness, compilability, coverage, or fault detection once code has been extracted, without explicitly assessing the reliability of that extraction step. Their evaluation pipelines therefore proceed directly from raw responses to downstream metrics, leaving output format and extractability unexamined. In this study we adapt the MSR/CSR framework of Macedo et al. (2024), originally proposed for code translation, to JUnit test generation, making format and extractability an explicit precondition before syntax and compilation analyses (RQ2).

Summary of **RQ1:** Prompt engineering strongly influences extractability of LLM-generated test suites. OpenAI models (GPT-3.5-turbo, GPT-4) achieve the highest MSR and CSR, with Mistral 7B competitive and Mixtral 8×7B lagging. GToT consistently yields the highest MSR across all models and datasets, enforcing structural adherence. For CSR, GToT often attains the best results (particularly for GPT models), though simpler prompts like ZSL or CoT occasionally match it depending on model–dataset pairing. FSL underperforms across all settings. These findings extend Macedo et al. (2024) by showing that controlling prompt structure remains crucial in test generation: explicit reasoning and delimiter guidance substantially enhance format compliance, yet no model follows delimiters reliably. Extracting usable test code from mixed natural-language outputs remains a key challenge for practical adoption.

4.3 [RQ2]: Syntactical Correctness and Compilability.

[Experiment Goal]: This research question examines how prompt engineering influences test suite syntactical correctness and compilability in LLM-generated tests. Valid Java grammar and compilation without modifications are essential for automated workflow

integration. We aim to identify the most effective prompting strategies for generating valid, executable test suites. To provide a reference point, we also report EvoSuite's syntactic correctness and compilability, as it represents a state-of-the-art SBST baseline. This contrast highlights differences between search-based and LLM-based approaches in producing directly usable test code.

[Experiment Design]: We evaluate syntactical correctness using Javalang¹³ parsing and assess compilability with JVM compilation. A test suite is considered syntactically correct if parsing succeeds, and compilable if compilation completes without errors. For non-compilable suites, we categorize compilation errors (e.g., `Cannot Find Symbol`, `Package Does Not Exist`) to identify hallucination patterns. We then analyze how different prompting techniques affect these error distributions and assess their effectiveness in mitigating specific hallucinations. EvoSuite results are included to indicate the upper bound expected from an SBST tool, allowing us to contextualize LLM outputs against a mature baseline. Experiments span SF110, Defects4J, and CMD datasets to evaluate how both paradigms handle correctness, compilability, and hallucination mitigation across diverse codebases.

Each run, whether by an LLM or by EvoSuite, produces exactly one test suite, which corresponds to a single Java test file. Therefore, all counts and percentages in our study are calculated on the basis of test files, ensuring a consistent comparison between LLM-generated and SBST-generated outputs.

For the calculation of syntactic correctness and compilability percentages, we proceed as follows. Suppose A is a given model and B a given dataset. Let X be the total number of syntactically correct test suites produced by (A, B) across all prompting strategies. For a specific prompting strategy p , we denote by Y_p the number of syntactically correct test suites it generates, and by Z_p the number of those that also compile successfully. The syntactic correctness percentage for (A, B, p) is then computed as $Y_p/X \times 100$, while the compilability percentage is computed as $Z_p/X \times 100$. For EvoSuite, which does not involve prompting strategies, syntactic correctness is measured as the fraction of syntactically correct test suites among all generated ones, and compilability is measured as the fraction of compilable test suites among those that are syntactically correct.

[Experiment Results]:

Dataset-Level Analysis.

Across all LLM runs, we produced **216,300** outputs (Table 3, Table 4). Of these, **190,619 (88.13%)** contained extractable test code, while the remaining **25,681 (11.87%)** were plain text or structurally invalid (no extractable code). Among the code-containing outputs, **171,187 (89.81%)** were syntactically correct, while **19,432 (10.19%)** exhibited syntax errors. For EvoSuite, all generated test files (**16,560** across Defects4J and SF110) were syntactically correct. These results highlight the challenge LLMs face in consistently generating well-formed unit tests, emphasizing the need for post-generation validation.

Finding 6: Across datasets, **89.81%** of LLM-generated outputs containing code are syntactically correct, while **10.19%** include syntax errors, requiring manual intervention before integration into real-world workflows. In contrast, EvoSuite achieves **100%** syntactic correctness on all its generated outputs.

¹³ <https://github.com/c2nes/javalang>

Table 7 Syntactic correctness (S) and compilability (C) by model, dataset, and prompt

Model	Dataset	ZSL (S/C)	FSL (S/C)	CoT (S/C)	ToT (S/C)	GToT (S/C)
GPT-3.5-turbo	Defects4J	12056/1656	9207/1244	9812/1808	11220/1268	13756/3789
	SF110	4541/1267	3689/897	3849/1164	4200/870	5042/2061
	CMD	916/114	877/103	848/250	692/113	889/146
GPT-4	CMD	894/69	890/86	804/103	684/74	615/93
Mistral 7B	Defects4J	11649/258	11633/247	11871/153	11575/252	10766/629
	SF110	4639/238	4617/227	4574/199	4460/379	3543/735
	CMD	792/0	789/1	801/0	782/1	724/12
Mixtral 8×7B	CMD	472/4	582/3	461/2	388/11	588/4
EvoSuite	Defects4J	11100/10890				
	SF110	5460/5460				

S = number of tests that parse successfully (syntactically correct); C = number of tests that compile without errors. Each cell reports S/C

To provide full transparency, Table 7 reports the raw counts of syntactically correct (S) and compilable (C) test suites for each model, dataset, and prompting strategy. These raw values form the basis for the normalized percentages of syntactic correctness and compilability later reported in Tables 8 and 9.

EvoSuite Compilation Outcomes. While EvoSuite achieved full syntactic correctness on all the test files it generated, a closer inspection reveals that not all of these tests compiled successfully. In particular, we identified 210 non-compilable cases across Defects4J: two in JacksonXml, two in Math, and three in JacksonDatabind. This represents a small fraction of the total (210 out of 11,100 syntactically correct suites, i.e., 1.9%), yet it highlights that even mature SBST tools can occasionally produce outputs that fail at compilation. On SF110, by contrast, EvoSuite compiled all 5,460 generated test files without error.

Finding 7: Even EvoSuite, the SBST baseline, does not guarantee full compilability: although all its generated tests were syntactically correct, 210 cases in Defects4J ($\approx 1.9\%$) failed to compile, while SF110 compiled without errors. This shows that compilation failures are not unique to LLMs but can also occur in mature automated test generation tools.

LLM and Prompting Impact on Syntactical Correctness. Table 8 reveals significant variations in syntactical correctness across models and prompting techniques. GPT-3.5-turbo with GToT achieves the highest correctness on Defects4J (**24.54%**) and SF110 (**23.65%**), showing the benefit of structured reasoning for syntax. On CMD, however, it performs slightly lower than ZSL (21.06% vs. **21.70%**). GPT-4 also reaches high scores on CMD with ZSL (**23.00%**). For Mistral 7B, FSL performs best on SF110 (**21.15%**), while CoT is highest on Defects4J (**20.65%**). These results suggest that prompting effectiveness for syntactical correctness varies across models and datasets, with no single strategy universally superior. FSL remains inconsistent overall, and Mixtral 8x7B shows the weakest performance despite isolated peaks (e.g., **23.60%** with GToT on CMD).

Table 8 Syntactic Correctness Across Datasets and Techniques*

Model Name	Syntactically Correct SF110 (%)					Syntactically Correct Defects4J (%)					Syntactically Correct CMD (%)				
	ZSL	FSL	CoT	ToT	GToT	ZSL	FSL	CoT	ToT	GToT	ZSL	FSL	CoT	ToT	GToT
GPT 3.5-turbo	21.30	17.30	18.05	19.70	23.65	21.51	16.43	17.51	20.02	24.54	21.70	20.77	20.09	16.39	21.06
GPT 4	-	-	-	-	-	-	-	-	-	-	23.00	22.90	20.68	17.60	15.82
Mistral 7B	21.25	21.15	20.95	20.43	16.23	20.26	20.23	20.65	20.13	18.73	20.37	20.29	20.60	20.11	18.62
Mixtral 8x7B	-	-	-	-	-	-	-	-	-	-	18.85	23.36	18.51	15.58	23.60
Average (%)	21.27	19.22	19.50	20.06	19.94	20.89	18.33	19.08	20.08	21.63	21	21.83	19.97	17.42	19.78
EvoSuite	100.00					100.00					-				

* Note that due to budget constraints, we only paid to query GPT-4 and Mixtral 8x7B for CMD

Table 9 Compilability Rates of Test Suites Across Datasets and Techniques*

Model Name	Compilable Code SF110 (%)				Compilable Code Defects4J (%)				Compilable Code CMD (%)						
	ZSL	FSL	CoT	ToT	ZSL	FSL	CoT	ToT	ZSL	FSL	CoT	ToT	ZSL	FSL	CoT
GPT 3.5-turbo	5.94	4.21	5.46	4.08	9.67	2.95	2.22	3.23	2.26	6.76	2.70	2.44	5.92	2.68	3.46
GPT 4	-	-	-	-	-	-	-	-	-	-	1.78	2.21	2.65	1.90	2.39
Mistral 7B	1.09	1.04	0.91	1.74	3.37	0.45	0.43	0.27	0.44	1.09	0.00	0.03	0.00	0.03	0.31
Mixtral 8x7B	-	-	-	-	-	-	-	-	-	-	0.16	0.12	0.08	0.44	20.16
Average (%)	3.52	2.62	3.19	2.91	6.52	1.70	1.32	1.75	1.35	3.93	1.16	1.20	2.16	1.26	1.58
EvoSuite	100.00	-	-	-	98.1	-	-	-	-	-	-	-	-	-	-

* ZSL: Zero-shot learning, FSL: Few-shot learning, CoT: Chain-of-Thought, ToT: Tree-of-Thoughts, GToT: Guided Tree-of-thoughts

Finding 8: GPT models outperform other LLMs in syntactic correctness, with GPT-4 achieving the highest correctness rate (23%) among all prompt configurations. Prompting effectiveness, however, is dataset-dependent: GToT yields top results on Defects4J and SF110, yet ZSL slightly surpasses it on CMD. No single prompting strategy proves universally superior, while Mixtral 8x7B remains the weakest performer overall despite occasional peaks (e.g., 23.6% with GToT on CMD).

Compilability Rates Across Datasets and Prompting Techniques. Table 9 reveals a significant gap between syntactical correctness and compilability. GPT-3.5-Turbo with GToT achieved the highest rates (9.67% on SF110, 6.76% on Defects4J), yet only 7.2% of generated tests and 12% of syntactically correct ones compiled. On CMD, GPT-3.5-Turbo with CoT surprisingly outperformed GPT-4 (5.92% vs. 2.65%), suggesting execution failures limit GPT-4's syntax advantage. Mixtral 8x7B showed inconsistent performance (near-zero to 20.16% with GToT), indicating occasional success but poor overall reliability.

These results contrast with prior work (Siddiq et al. 2024a), which reports substantially higher compilation rates. A key reason lies in the different experimental setups. Siddiq et al. (2024a) adopt a method-level perspective, generating one test class per method under test, with all necessary imports explicitly included and post-processing heuristics applied to repair non-compilable outputs. In contrast, our study operates at the class level, where each generation targets the full class under test, often containing multiple interdependent methods and relying solely on the code available in the dataset without import injection or post-repair. Furthermore, our benchmarks (Defects4J, CMD, SF110) encompass larger and more heterogeneous projects than HumanEval or the filtered SF110 subset used by Siddiq et al. (2024a), which further contributes to differences in compilation outcomes.

Finding 9: The gap between syntactical correctness and compilability remains substantial, only 7.2% of all generated test suites and 12% of syntactically correct ones successfully compile. Compared to prior work, which reports higher compilability when generating one test class per individual method under test with injected imports and repair heuristics, our study evaluates entire classes under test (including all their interdependent methods) without import injection or post-repair. This setup highlights the additional challenges LLMs face when handling complete classes rather than isolated methods.

LLMs vs. EvoSuite on Syntactic Correctness and Compilability. Tables 8 and 9 allow a direct comparison between LLMs and EvoSuite. EvoSuite exhibits near-perfect reliability, with all generated test suites being syntactically correct and only 210 non-compilable cases out of 11,100 in Defects4J ($\approx 1.9\%$), while achieving 100% compilability on SF110. By contrast, LLMs display a much higher fragility: across datasets, only 89.81% of code-containing outputs were syntactically correct, and compilability drops drastically, with success rates below 10% in most cases. Reasoning-based prompting (e.g., GToT) mitigates syntax issues and improves compilability, but still falls far short of EvoSuite's stability. This contrast underscores that while LLMs can approach EvoSuite in raw generation capacity (RQ0), their outputs are substantially less reliable when measured by structural validity and executable quality.

Finding 10: Compared to EvoSuite, LLMs remain substantially weaker in syntactic correctness and compilability: while EvoSuite compiles nearly all its generated tests (with only $\approx 1.9\%$ failures in Defects4J), LLMs succeed on fewer than 10% of their syntactically correct outputs. This highlights that prompt engineering improves LLM reliability but does not yet close the gap with mature SBST tools.

Compilation Error Analysis and Hallucination Patterns. Table 10 reveals dataset-specific hallucination patterns, with CMD showing highest rates. 'Cannot Find Symbol' (CFS) errors dominate across models: 86.47% (GPT-3.5-Turbo), 76.99% (GPT-4), 72.41% (Mistral 7B), and 67.91% (Mixtral 8x7B), indicating generation of non-existent symbols. 'Package Does Not Exist' (PDNE) errors are significant in Mixtral 8x7B (22.35%) and Mistral 7B (23.13%), reflecting incorrect imports.

Defects4J reveals encapsulation issues with high CFS (GPT-3.5: 80.57%, Mistral 7B: 82.12%) and PAI errors (GPT-3.5: 7.88%, Mistral 7B: 4.74%), suggesting access permission hallucinations. SF110 shows redundancy problems with prevalent CFS (GPT-3.5: 81.02%, Mistral 7B: 82.42%) and PAI peaking at 11.76% with GPT-3.5 Few-shot. PCR and DCD errors indicate LLMs generate redundant test structures. Our findings confirm Zhang et al. (2024b)'s conclusions that LLM hallucinations occur due to missing context and poor dependency handling, while extending their work by demonstrating these same patterns specifically in test generation contexts. The prevalence of CFS errors (up to 86.47% in GPT-3.5-Turbo) confirms their observation about fabricated references, while PDNE errors (up to 23.13% in Mistral 7B) extend their findings by revealing the particular challenges of dependency resolution in test suite generation.

Finding 11: LLM hallucinations are a major source of compilation errors. CMD is dominated by hallucinated missing symbols, Defects4J struggles with encapsulation violations, and SF110 suffers from redundant test generation.

Prompt Engineering and Hallucination Mitigation.

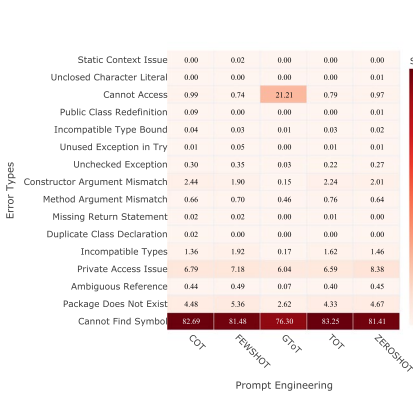
Figure 5 reveals the distribution of non-zero error types across different machine learning models and prompt engineering techniques, with the y-axis dynamically adapting to display error types specific to each model and prompt configuration. The heatmap (Fig. 5) shows CoT and GToT reduce hallucinations more effectively than ZSL and FSL. CoT lowers 'Private Access Issue' (PAI) errors to 2.96% (GPT-3.5) and 1.50% (GPT-4) in CMD, improving encapsulation adherence. GToT nearly eliminates errors in Defects4J through structured dependencies. However, hallucinations in symbol resolution and redundant test generation persist despite structured reasoning.

FSL increases PAI errors to 11.31% (GPT-3.5) and 5.64% (Mistral 7B) in SF110, showing persistent access restriction misinterpretations. 'Cannot Find Symbol' (CFS) errors remain high (86.47% in CMD, 79.16% in SF110), indicating examples don't resolve symbol hallucinations. ZSL performs worst with CFS errors reaching 86.47% (CMD) and 78.89% (SF110) due to minimal context. Our findings extend Zhang et al. (2024b, 2025) by demonstrating how reasoning-based prompting like CoT and GToT can reduce certain error types—particularly PAI and CAM—by reinforcing syntactic scaffolding. While reasoning-based prompting reduces errors, challenges in reference resolution, encapsulation, and redundant test generation remain.

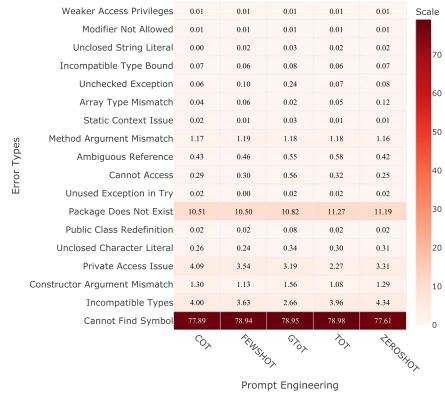
Table 10 Compilation Errors by Model and Dataset (All Prompting Techniques)

Model	Dataset	AR	ATM	CA	CFS	CR	CAM	DCD	ITB	IT	MAM	MRS	MNA	PDNE	PAI	PCR	SCI	UE	UCL	USL	UET	VNI	WAP
GPT 3.5-Turbo	Defects4J	0.45	0.0	0.83	84.21	0	2.26	0.0	0.04	1.58	0.24	0.01	0	3.53	6.36	0.02	0	0.43	0.01	0	0.04	0	0
	SF110	0.45	0.0	9.05	72.40	0	2.53	0.01	0.01	1.83	0.31	0.01	0	1.48	11.67	0.04	0.01	0.19	0	0	0	0	0
	CMD	0.21	0	0	86.48	0	0.46	0	0.03	0.51	1.39	0	0.00	7.87	2.96	0.00	0.00	0.08	0.00	0	0.01	0.00	0.00
GPT 4	CMD	0.21	0	0	76.99	0	0.24	0	0.05	1.08	3.03	0	0.00	16.46	1.50	0.11	0.01	0.30	0.01	0	0.00	0.01	0.00
	Defects4J	0.31	0.06	0.34	81.03	0.01	1.27	0.0	0.10	5.11	0.73	0.00	0.01	6.63	3.68	0.04	0.02	0.17	0.43	0.03	0.02	0	0
Mistral 7B	SF110	0.86	0.06	0.35	82.0	0	1.84	0.0	0.01	5.18	0.97	0.00	0	2.81	5.56	0.02	0.03	0.16	0.15	0.01	0	0	0
	CMD	0.29	0	0	72.41	0	0.71	0	0.09	0.86	1.82	0.02	23.13	0.61	0.04	0.00	0.00	0.00	0.00	0	0.00	0.00	0.01
Mixtral 8x7B	CMD	0.31	0	0	67.91	0	0.76	0	0.38	1.59	4.77	0.00	22.35	1.81	0.04	0.02	0.05	0.00	0.00	0	0.00	0.00	0.00

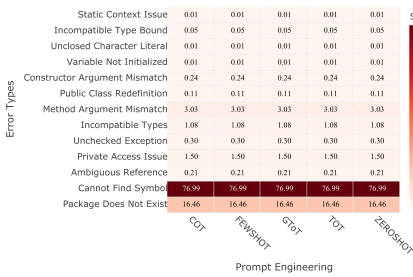
* AR: Ambiguous Reference, ATM: Array Type Mismatch, CA: Cannot Access, CFS: Cannot Find Symbol, CR: Class Redefinition, CAM: Constructor Argument Mismatch, DCD: Duplicate Class Declaration, ITB: Incompatible Type Bound, IT: Incompatible Types, MAM: Method Argument Mismatch, MRS: Missing Return Statement, MNA: Modifier Not Allowed, PDNE: Package Does Not Exist, PAI: Private Access Issue, PCR: Public Class Redefinition, SCI: Static Context Issue, UE: Unchecked Exception, UCL: Unclosed Character Literal, USL: Unclosed String Literal, UET: Unused Exception in Try, VNI: Variable Not Initialized, WAP: Weaker Access Privileges



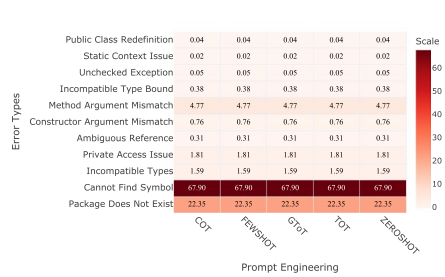
(a) By GPT 3.5-Turbo and by prompt



(b) By Mistral 7B and by prompt engineering



(c) By GPT 4 and by prompt



(d) By Mixtral 8x7B and by prompt engineering

Fig. 5 Heatmap of compilation errors by model and prompt engineering

Finding 12: Reasoning-based prompting (CoT, GToT) significantly reduces errors, particularly Private Access Issues, whereas Zero-shot Learning exacerbates compilation failures by generating unstructured, non-compilable test cases.

Relation to Prior Work Prior studies report higher LLM test compilability, but typically use configurations that simplify the task through granularity restrictions, import handling, or repair. Siddiq et al. (2024a) generate one class per focal method and apply deterministic clean-up heuristics (stripping prose, fixing packages, bracket trimming), substantially inflating post-fix compilability; their raw SF110 compilability is very low with coverage under 2% without repairs. Yang et al. (2024) normalize imports and recursively prune failing test methods until classes compile, measuring coverage only on compilable residue; they attribute low validity to unresolved symbols, parameter mismatches, and abstract instantiation. Tang et al. (2024) achieve 70% runnable tests after light IDE fixes with a single prompt, yet EvoSuite dominates coverage. Yuan et al. (2024) document modest raw ChatGPT compilability (57.9% compilation errors) but large gains with iterative compile-feedback (ChatTester). We evaluate full classes without import injection or post-hoc repair, yielding lower compilability but exposing true failure mode distributions—including occasional EvoSuite failures on Defects4J. Our RQ2 confirms and refines prior

taxonomies from Yang et al. (2024): beyond unresolved symbols and abstract instantiation, we distinguish cannot-find-symbol, missing-package, private-access, and redundant-constructors, quantifying how their prevalence shifts across CMD, Defects4J, and SF110. Uniquely, we analyze these errors under reasoning-oriented prompting, observing that such prompts reduce specific error classes (e.g., private-access violations) while leaving others (e.g., unresolved symbols) largely unaffected. Our findings extend broader code-generation hallucination accounts to unit-test generation, showing prompt design can selectively mitigate validity failures.

Summary of **RQ2**: LLMs generate a high proportion of syntactically correct tests ($\approx 90\%$), yet compilability remains low ($< 10\%$) due to unresolved symbols, missing packages, and access violations. EvoSuite, while more reliable, also exhibits rare compilation failures ($\approx 1.9\%$ in Defects4J), showing that errors are not unique to LLMs. Prompt engineering plays a critical role: reasoning-based prompts (CoT, GToT) improve structural correctness and reduce certain hallucinations, whereas FSL consistently underperforms. Compared to prior studies that operate at the method level with imports and repairs, our class-level, no-repair setup highlights the additional challenges of generating fully compilable suites directly from raw LLM outputs.

4.4 [RQ3]: Static Analysis of Human-Oriented Quality

[Experiment Goal]: This research question evaluates the quality of LLM-generated test suites from a static analysis perspective. Unlike EvoSuite, which primarily targets structural coverage, here the focus is on human-oriented dimensions such as coding standard compliance, maintainability (via cyclomatic and cognitive complexity), and potential defects. By analyzing these aspects, we assess whether prompt design can improve the readability and long-term usability of automatically generated tests.

[Experiment Design]: We evaluate test quality using three approaches. Checkstyle assesses adherence to Google Java Style Checkstyle and Sun Code Conventions, measuring formatting consistency and standard violations. PMD analyzes cyclomatic complexity (independent paths) and cognitive complexity (comprehension ease) (Dantas and Maia 2021). SpotBugs detects potential defects categorized by severity: Scariest (critical issues), Scary (high-risk errors), Troubling (moderate concerns), and Of Concern (minor warnings). We compared prompting techniques (ZSL, FSL, CoT, ToT, GToT) across SF110, Defects4J, and CMD datasets.

[Experiment Results]:

Coding Standard Compliance. LLM-generated test suites show varying coding standard adherence (Fig. 6). FSL achieves highest compliance with Google Java Style and Sun Code Conventions, producing fewer violations overall. For Google Java Style (Fig. 6(a)), indentation violations dominate across all prompting techniques, with GToT showing the lowest count (1757.5 vs. 2305.9 for ZSL). However, GToT increases import-related violations (140.6 CustomImportOrder and 93.1 AvoidStarImport violations) compared to FSL (71.3 and 23.4 respectively). For Sun Code Conventions (Fig. 6(b)), MissingJavadocMethod violations are highest with CoT (566.5) and GToT (564.0), while GToT significantly reduces MagicNumber violations (134.8 vs. 323.4 for ZSL). These patterns suggest that reasoning-based prompts (CoT, ToT, GToT) improve code structure but sacrifice documentation and

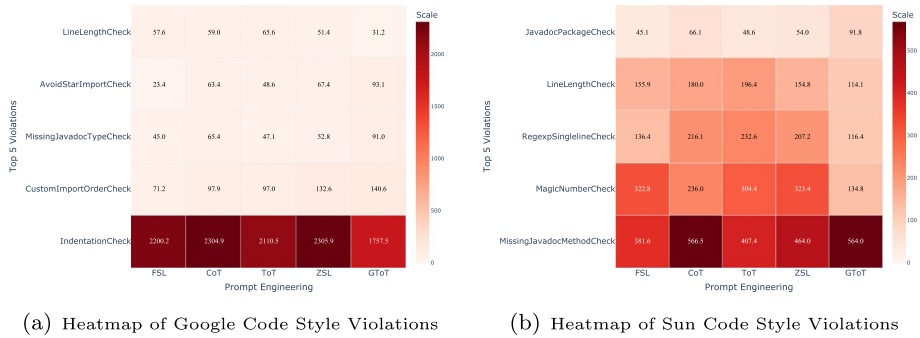


Fig. 6 Heatmap of the Top-5 Google and Sun Code Style Violations Across Prompt Engineering

import organization. FSL strikes the best balance, minimizing both structural and documentation violations. These findings partially align with Tang et al. (2024), who reported high ZSL violations, though our results show FSL and CoT significantly improve compliance while ToT and GToT produce tests with better logical structure but less style adherence.

Finding 13: FSL achieves the highest coding standard compliance, followed by CoT, which also reduces violations and improves adherence to style guidelines. In contrast, GToT enhances logical organization but introduces more violations, reinforcing the trade-off between structured prompting and strict adherence to coding conventions.

Code Complexity Analysis. Table 11 shows that all prompting strategies produce test suites with very low structural complexity: more than 99.5% of outputs fall into the lowest cyclomatic and cognitive complexity bands. GToT shows slightly higher proportions of simple cases (99.99% cyclomatic, 100% cognitive) and is the only strategy that avoids all moderate or high cases in cognitive complexity while nearly eliminating them in cyclomatic complexity (0.01% moderate, 0% high). Although these differences are marginal in absolute terms, they point to a stabilizing effect of GToT, reducing structural anomalies rather than changing the overall average complexity.

Table 11 Summary of Complexity Results by Prompt Engineering Techniques

Complexity Metric	ZSL	FSL	CoT	ToT	GToT
(a) Cyclomatic Complexity Results					
Low (1-4)	99.86	99.50	99.73	99.73	99.99
Moderate (5-7)	0.11	0.39	0.16	0.20	0.01
High (8-10)	0.03	0.11	0.11	0.07	0
Very High (11+)	0	0	0	0	0
(b) Cognitive Complexity Results					
Low (< 5)	99.83	99.76	99.76	99.90	100
Moderate (5-10)	0.11	0.14	0.14	0.03	0
High (11-20)	0.03	0.10	0.11	0.07	0
Very High (21+)	0.03	0	0	0	0

ZSL: Zero-shot learning, FSL: Few-shot learning, CoT: Chain-of-Thought, ToT: Tree-of-Thoughts, GToT: Guided Tree-of-Thoughts

These findings refine the observations of Tang et al. (2024), who showed that ZSL already produces simple tests, by highlighting that reasoning-based prompting can maintain this simplicity while reducing occasional outlier cases.

Finding 14: All prompting strategies yield predominantly low-complexity test suites. GToT achieves slightly higher proportions of simple outputs and is the only strategy with zero moderate or high cases in cognitive complexity, and nearly none in cyclomatic complexity. While differences across strategies are small, GToT appears to offer more structural consistency by reducing complexity outliers.

Static Analysis with SpotBugs.

Table 12 reports the distribution of static analysis warnings (SpotBugs) across 20,530 compilable test suites. While most defects are classified as minor “Of Concern” (55.6%), we observe notable differences across prompting strategies. FSL produces the most “Scariest” defects (e.g., 410 on SF110), while GToT yields the highest number of “Scary” (825) and “Troubling” (825) issues, particularly on SF110. These results highlight a concerning trade-off: although GToT improves structure and readability, it tends to introduce more subtle but structurally embedded defects. ZSL, while producing simpler tests, yields far fewer severe warnings. Unlike prior work (Tang et al. 2024), which focused on ZSL with limited compilation filtering, our evaluation reveals how reasoning-oriented prompting can unintentionally lead to more severe code issues, emphasizing the need for robust post-generation validation.

Table 12 Bug Priority Levels by Prompt Techniques

Prompt	Bug Level	GPT-3.5			GPT-4	Mistral7B			Mistral8x7B
		SF110	D4J	CMD	CMD	SF110	D4J	CMD	CMD
ZSL	Of Concern	600	212	17	15	129	48	0	2
	Scariest	0	0	0	0	129	48	0	0
	Scary	0	0	0	0	129	0	0	0
	Troubling	0	212	0	15	0	0	0	0
FSL	Of Concern	410	177	18	14	135	60	0	0
	Scariest	410	177	0	0	135	60	0	0
	Scary	0	0	0	0	0	0	0	0
	Troubling	0	0	0	14	0	0	0	0
CoT	Of Concern	553	219	22	0	125	32	0	0
	Scariest	0	0	0	0	0	32	0	0
	Scary	0	0	0	0	125	0	0	0
	Troubling	0	0	0	0	125	0	0	0
ToT	Of Concern	20	21	3	7	212	63	1	0
	Scariest	0	1	0	0	212	63	0	0
	Scary	1	1	0	0	212	0	0	0
	Troubling	0	2	0	1	0	0	0	0
GToT	Of Concern	825	242	20	19	347	131	0	4
	Scariest	0	0	0	0	0	0	0	0
	Scary	825	0	0	0	0	0	0	0
	Troubling	825	0	0	0	0	0	0	0

* ZSL = Zero-shot Learning, FSL = Few-shot Learning, CoT = Chain-of-Thought, ToT = Tree-of-Thought, GToT = Guided Tree-of-Thought. SpotBugs ranks issue severity as follows: **Scariest** (most critical), **Scary**, **Troubling**, and **Of Concern** (least critical)

Finding 15: Prompting strategies impact bug prevalence differently. FSL introduces the most critical (“Scariest”) issues, while GToT yields the highest number of structurally embedded “Scary” and “Troubling” bugs, particularly on SF110. ZSL produces the fewest severe warnings, albeit with simpler outputs. These results call for stricter validation when using reasoning-based prompting.

Relation to Prior Work Tang et al. (2024) analyze human-oriented quality through Checkstyle, PMD, and SpotBugs, showing that ChatGPT-generated tests under zero-shot prompting often violate style guidelines but remain structurally simple, with low cyclomatic and cognitive complexity. Yuan et al. (2024) complement this perspective with a user study, where developers judged LLM-generated tests comparable to manual ones in terms of usability and adoption effort. By contrast, neither Yang et al. (2024) nor Siddiq et al. (2024a) evaluate readability or maintainability, focusing instead on correctness and coverage. Our study extends this line of work by systematically comparing prompting strategies (ZSL, FSL, CoT, ToT, GToT) and showing how they shift coding standard compliance, complexity, and defect incidence. Unlike prior studies limited to zero-shot prompting, we demonstrate that reasoning-based prompts can improve logical organization at the cost of higher defect density, while FSL maximizes style compliance. This nuanced trade-off analysis situates prompt engineering as a key determinant of human-oriented quality in LLM-generated tests, beyond what was captured in earlier evaluations.

Summary of **RQ3:** LLM-generated tests generally exhibit low complexity and readable structures, consistent with prior reports. However, coding-standard compliance and defect incidence vary strongly with prompting strategies. FSL yields the cleanest style adherence, while reasoning-based prompts (CoT, ToT, GToT) enhance logical organization but introduce more subtle bugs flagged by SpotBugs. Compared to prior work limited to zero-shot prompting, our analysis shows that prompt engineering shifts the balance between readability, maintainability, and defect-proneness, underscoring its central role in shaping human-oriented quality.

4.5 [RQ4]: Readability of LLM-Generated Test Suites: Alignment with Developer Expectations

[Experiment Goal]: This experiment investigates whether LLM-generated test suites meet developer expectations of readability, focusing on clarity, structure, and ease of navigation. We use the readability model by Scalabrino et al. (2018), which has been validated against human judgments, to quantify readability beyond surface-level formatting. EvoSuite is included here as a baseline reference: although it does not explicitly optimize for readability, its widespread use in SBST makes it a valuable point of comparison to assess whether LLM prompting strategies deliver tangible improvements in human-oriented quality.

[Experiment Design]: We employ the readability model of Scalabrino et al. (2018), trained on Java code including JUnit tests and validated against human judgments. The model evaluates syntactic structure, identifier clarity, textual coherence, and overall structural readability, thus going beyond superficial formatting checks. All analyses are restricted to compilable test suites, ensuring that readability is measured on executable outputs. The evaluation is conducted in three complementary stages. First, a dataset-wise

analysis: we aggregate readability scores by dataset (Defects4J, SF110, CMD) for GPT-3.5-turbo and Mistral 7B. EvoSuite is also included on Defects4J and SF110, where it was executable, but excluded from CMD due to Java version incompatibilities (Section 3.6). This view highlights how readability varies with dataset characteristics. Second, a prompt-wise analysis: we aggregate results by prompting strategy (ZSL, FSL, CoT, ToT, GToT) for GPT-3.5-turbo and Mistral 7B, and contrast them with EvoSuite on Defects4J and SF110. Although EvoSuite does not explicitly optimize for readability, its inclusion provides a widely used SBST baseline against which to assess the added value of prompting strategies. Finally, for CMD we restrict the analysis to LLMs only, since EvoSuite could not be executed due to Java version incompatibilities. All LLMs are evaluated across the five prompting strategies, but readability scores are reported only for configurations that yielded compilable test suites. For every configuration (dataset-wise, prompt-wise, CMD-only), readability distributions are summarized using descriptive statistics (minimum, quartiles, median, maximum, mean). Higher scores indicate more structured, navigable, and human-readable test suites.

[Experiment Results]:

Dataset-Level Readability Analysis.

Table 13 shows LLM-generated test suites consistently achieve higher readability than EvoSuite across all datasets. Readability varies with codebase complexity: Defects4J exhibits highest readability scores (GPT-3.5-Turbo: 81.73%, Mistral 7B: 69.06% vs. EvoSuite: 57.33%), followed by SF110 (GPT-3.5-Turbo: 75.95%, Mistral 7B: 70.28% vs. EvoSuite: 50.67%). CMD presents lowest scores (GPT-3.5-Turbo: 60.93%, Mistral 7B: 57.13%), indicating modern, evolving codebases challenge LLMs more. While LLMs consistently outperform EvoSuite in readability, their effectiveness varies with dataset complexity.

Finding 16: LLM-generated tests outperform EvoSuite in readability. However, readability varies with dataset complexity, as shown in Table 13. CMD, which contains modern and evolving codebases, presents the greatest challenge, yielding the lowest readability scores across all models.

LLM vs. EvoSuite: Readability Performance.

Table 14 shows LLM-generated tests consistently surpass EvoSuite in readability. For GPT-3.5-Turbo, GToT achieves the highest score (84.75% vs. EvoSuite's 50.77%), followed by CoT (82.77%) and ToT (73.32%), while ZSL (77.93%) and FSL (73.49%) score lower. Similarly, for Mistral 7B, GToT leads (76.35%), followed by ToT (69.64%), with ZSL (63.79%) and FSL (60.40%) producing lowest scores. These results demonstrate reasoning-based prompting's importance for readability.

Table 13 Statistical Metrics from Readability Score Distribution by Model and Dataset

Model	Dataset	Min	Q1	Median	Q3	Max	Mean
GPT 3.5-turbo	Defects4J	33.35	70.53	84.03	95.74	99.04	81.73
	SF110	25.02	66.32	76.12	87.10	98.64	75.95
	CMD	41.34	55.53	61.30	65.98	90.45	60.93
Mistral7B	Defects4J	15.20	61.23	68.44	78.03	97.58	69.06
	SF110	6.58	60.19	71.04	80.99	98.59	70.28
	CMD	13.60	38.68	52.68	84.24	95.07	57.13
EvoSuite	Defects4J	16.10	44.67	56.04	72.04	95.70	57.33
	SF110	16.33	36.83	50.03	61.81	89.67	50.67

Notice. Values are aggregated over all prompting strategies that produced compilable test suites

Table 14 Statistical Metrics from Readability Score Distribution (Defects4J and SF110)

Prompt	Min	Q1	Median	Q3	Max	Mean
(a) GPT 3.5-turbo by Prompt vs EvoSuite						
ZSL	29.37	66.79	77.50	93.50	98.41	77.93
FSL	25.02	60.24	74.36	88.07	97.78	73.49
CoT	43.28	73.17	85.74	95.33	99.04	82.77
ToT	33.36	64.33	72.04	82.64	97.83	73.32
GToT	42.24	75.20	89.39	96.26	98.64	84.75
EvoSuite	4.11	36.96	51.02	62.59	97.64	50.77
(b) Mistral7B by Prompt vs EvoSuite						
ZSL	25.67	57.52	64.41	71.87	90.62	63.79
FSL	6.58	49.03	62.53	70.78	97.58	60.40
CoT	34.54	57.67	65.39	73.92	97.77	65.92
ToT	33.30	60.94	69.00	78.29	97.67	69.64
GToT	15.20	67.47	77.24	89.02	98.59	76.35
EvoSuite	4.11	36.96	51.02	62.59	97.64	50.77

Finding 17: LLM-generated test suites improve readability over EvoSuite by 21–40%, as shown in Table 14. This highlights the significant advantage of LLMs in producing more comprehensible test code.

Prompting Strategies: Influence on Readability.

Table 15 shows reasoning-based prompting significantly enhances test suite clarity. GToT and CoT consistently achieve highest readability scores across datasets. For GPT-3.5-Turbo on CMD, CoT leads (67.99%), followed by GToT (61.86%), while ZSL (57.67%) and FSL (55.76%) score lower. Similarly, for GPT-4 on CMD, GToT achieves highest readability (67.71%). These results confirm minimal prompting provides insufficient guidance for well-structured tests, while reasoning-based approaches are essential for producing maintainable test suites.

Finding 18: CoT and GToT consistently enhance readability, even on complex datasets like CMD, where unstructured prompting results in lower scores.

Model-Specific Readability Trends.

Table 15 reveals how different LLMs respond to reasoning-based prompting. GPT-3.5-Turbo performs best with CoT (67.99%), while GPT-4 excels with GToT (67.71%), confirming larger models benefit from structured reasoning. Mistral 7B shows higher variability, reaching 64.61% with GToT, indicating sensitivity to prompt design. Mixtral 8x7B performs exceptionally with ToT (79.93%), though its limited sample size affects stability.

Finding 19: Model architecture, parameter size, and prompting strategy significantly impact readability, with larger models consistently improving when guided by reasoning-based prompting.

Table 15 Statistical Metrics from Readability Score Distribution on CMD

Prompt	Min	Q1	Median	Q3	Max	Mean
(a) GPT 3.5-turbo on CMD						
ZSL	41.34	53.19	57.95	63.26	69.17	57.67
FSL	42.30	49.14	55.80	61.70	66.07	55.76
CoT	45.92	62.59	66.37	71.95	90.45	67.99
ToT	47.74	57.77	62.09	64.61	71.10	61.38
GToT	52.07	57.29	61.02	67.15	70.12	61.86
(b) GPT 4 on CMD						
ZSL	36.69	47.61	50.20	57.81	71.66	52.13
FSL	30.14	47.15	50.56	56.03	71.80	51.62
CoT	51.51	57.83	61.42	66.12	75.20	61.64
ToT	54.02	59.94	62.86	66.51	73.04	63.18
GToT	58.10	63.35	68.85	71.08	76.13	67.71
(c) Mistral7B on CMD						
ZSL	–	–	–	–	–	–
FSL	35.18	35.18	35.18	35.18	35.18	35.18
CoT	–	–	–	–	–	–
ToT	49.18	49.18	49.18	49.18	49.18	49.18
GToT	13.60	45.53	74.89	93.97	95.07	64.61
(d) Mixtral8x7B on CMD						
ZSL	38.43	44.17	49.91	55.65	61.39	49.91
FSL	48.36	48.36	48.36	48.36	48.36	48.36
CoT	46.46	51.88	57.29	62.71	68.12	57.29
ToT	57.14	77.90	79.08	92.66	92.85	79.93
GToT	44.32	58.67	75.75	89.30	93.00	72.21

* Since only compilable code was considered, there was no compilable code for Mistral 7B's CoT and ZSL prompts

Relation to Prior Work Readability from a developer perspective has been explicitly studied by Yuan et al. (2024), who conducted a user study showing that ChatGPT-generated tests were rated comparable to manual ones in terms of readability and adoption effort. Other evaluations, such as Tang et al. (2024), focus on understandability proxies (e.g., style violations, complexity) rather than direct readability. Our study extends this line of work by systematically quantifying readability with an established model validated against human judgments, and by comparing LLMs with EvoSuite. We further show how prompting strategies (CoT, ToT, GToT) significantly enhance readability beyond zero-shot and few-shot settings, demonstrating that structured reasoning prompts align LLM outputs more closely with developer expectations.

Summary of **RQ4**: LLM-generated tests consistently achieve higher readability than EvoSuite, with gains of 21–40% across datasets. Readability varies with dataset complexity, being highest on Defects4J and lowest on CMD. Reasoning-based prompting (CoT, GToT) markedly improves clarity and navigability, while zero-shot and few-shot remain weaker. Compared to prior work, which assessed readability through targeted developer studies, our study provides a systematic, model-based evaluation showing that prompting strategies play a central role in aligning LLM outputs with developer expectations.

4.6 [RQ5]: Code Coverage.

[Experiment Goal]: This experiment evaluates LLM-generated test suite effectiveness in achieving structural coverage. We measure line, instruction, and method coverage, comparing LLMs against EvoSuite and examining how prompting strategies impact coverage. Method coverage is particularly important in our class-level evaluation to determine cross-method interaction rather than isolated functionality testing.

[Experiment Design]: We use JaCoCo¹⁴ to measure line coverage (executed source lines), instruction coverage (executed bytecode instructions), and method coverage (methods invoked at least once). Following Tang et al. (2024), branch coverage is excluded as it focuses only on conditional branches and often yields missing values. Coverage is computed exclusively on compilable test suites (for LLMs: code that parsed and compiled; for EvoSuite: compiled outputs only). To make comparisons both fair and informative, we report coverage along two complementary views. First, a prompt-wise view on the common dataset pool: for GPT-3.5-turbo and Mistral 7B, coverage distributions produced by each prompting strategy (ZSL, FSL, CoT, ToT, GToT) are aggregated over SF110 and Defects4J—the datasets where EvoSuite is available—and contrasted with the single EvoSuite baseline on the same pool. (EvoSuite has no prompting; it serves as a fixed reference distribution.) Second, a dataset-wise view for LLMs: we provide per-dataset coverage distributions on SF110, Defects4J, and CMD to expose context sensitivity across benchmarks. Model-dataset availability follows our experimental constraints (Section 3.6). GPT-3.5-turbo and Mistral 7B were run on all three datasets, whereas GPT-4 and Mixtral 8×7B were restricted to CMD due to budget constraints. EvoSuite could not be executed on CMD because of Java version incompatibilities. Consequently, CMD analyses are LLM-only; within CMD, we report prompt-wise results for GPT-3.5-turbo and GPT-4. Mixtral 8×7B is omitted from CMD coverage summaries due to negligible compilability, yielding insufficient samples for meaningful statistics. For every configuration (prompt-wise or dataset-wise), we summarize the resulting coverage distributions using standard descriptive statistics (minimum, quartiles, median, maximum, mean) to capture both central tendency and dispersion.

[Experiment Results]:

EvoSuite vs. LLMs: Overall Coverage Performance.

Tables 16, 17, and 18 reveal a significant performance gap between LLMs and EvoSuite. EvoSuite consistently outperforms LLMs across all metrics, achieving median coverage of 94.34% (line), 95.0% (instruction), and 100% (method) through search-based optimization. LLM-generated tests show higher variability, with reasoning-based prompting significantly improving coverage over ZSL and FSL. GPT-3.5-Turbo with CoT achieves the highest LLM median coverage (75.47% line, 76.0% instruction, 100% method), though all LLM configurations fall short of EvoSuite’s systematic exploration capabilities.

Finding 20: EvoSuite outperforms all LLMs in coverage metrics. However, reasoning-based techniques (CoT, GToT, ToT) significantly enhance LLM test effectiveness, reducing the performance gap.

¹⁴<https://www.jacoco.org/jacoco/>

Table 16 Statistical metrics from Line coverage distribution

Prompt	Min	Q1	Median	Q3	Max	Mean
(a) EvoSuite vs GPT 3.5-turbo.						
ZSL	0.0	1.49	27.68	100.0	100.0	46.3
FSL	0.0	1.73	16.14	100.0	100.0	41.2
CoT	0.0	3.33	75.47	100.0	100.0	55.56
ToT	0.0	1.82	24.39	93.55	100.0	41.96
GToT	0.0	4.65	41.54	100.0	100.0	46.7
EvoSuite	0.0	88.61	94.34	97.67	100.0	91.75
Prompt	Min	Q1	Median	Q3	Max	Mean
(b) EvoSuite vs Mistral 7B.						
ZSL	0.0	0.0	9.2	67.27	100.0	31.51
FSL	0.0	1.97	18.45	63.04	100.0	31.58
CoT	0.0	0.0	1.12	65.22	100.0	30.28
ToT	0.0	1.2	20.0	76.44	100.0	37.0
GToT	0.0	4.55	30.0	83.89	100.0	42.41
EvoSuite	0.0	88.61	94.34	97.67	100.0	91.75
Dataset	Min	Q1	Median	Q3	Max	Mean
(c) GPT 3.5-turbo by datasets.						
Defects4J	0.0	10.22	82.74	100.0	100.0	62.61
SF110	0.0	0.0	3.57	23.24	100.0	22.72
CMD	4.0	53.59	100.0	100.0	100.0	76.86
Dataset	Min	Q1	Median	Q3	Max	Mean
(d) Mistral 7B by datasets.						
Defects4J	0.0	4.38	63.4	89.47	100.0	52.55
SF110	0.0	0.0	3.03	25.0	100.0	15.15
CMD	0.0	0.0	0.0	0.0	0.0	0.0
Prompt	Min	Q1	Median	Q3	Max	Mean
(e) GPT 3.5-turbo on CMD.						
ZSL	4.0	22.88	61.29	92.31	100.0	56.05
FSL	8.33	30.0	83.33	100.0	100.0	67.78
CoT	8.33	92.05	100.0	100.0	100.0	84.53
ToT	31.82	73.39	100.0	100.0	100.0	85.84
GToT	31.82	67.8	100.0	100.0	100.0	84.27
Prompt	Min	Q1	Median	Q3	Max	Mean
(f) GPT 4 on CMD.						
ZSL	0.0	0.0	8.33	70.97	89.47	31.88
FSL	5.56	12.89	52.51	85.94	100.0	52.74
CoT	0.0	18.68	65.22	100.0	100.0	60.35
ToT	0.0	28.25	72.22	100.0	100.0	63.82
GToT	0.0	31.61	72.22	100.0	100.0	64.01

Table 17 Statistical metrics from Instruction coverage distribution

Prompt	Min	Q1	Median	Q3	Max	Mean
(a) EvoSuite vs GPT 3.5-turbo.						
ZSL	0.0	1.0	29.0	100.0	100.0	46.26
FSL	0.0	1.0	14.0	100.0	100.0	41.72
CoT	0.0	3.0	76.0	100.0	100.0	55.64
ToT	0.0	1.0	27.0	96.0	100.0	42.91
GToT	0.0	3.5	45.0	100.0	100.0	47.61
EvoSuite	0.0	91.0	95.0	98.0	100.0	93.35
Prompt	Min	Q1	Median	Q3	Max	Mean
(b) EvoSuite vs Mistral 7B.						
ZSL	0.0	0.0	10.0	73.25	100.0	33.63
FSL	0.0	1.0	18.0	60.5	100.0	32.48
CoT	0.0	0.0	0.0	83.0	100.0	31.19
ToT	0.0	0.0	17.0	81.0	100.0	37.49
GToT	0.0	3.25	26.0	87.25	100.0	42.31
EvoSuite	0.0	91.0	95.0	98.0	100.0	93.35
Dataset	Min	Q1	Median	Q3	Max	Mean
(c) GPT 3.5-turbo by datasets.						
Defects4J	0.0	11.0	85.0	100.0	100.0	63.14
SF110	0.0	0.0	2.0	24.5	100.0	22.58
CMD	2.0	48.25	100.0	100.0	100.0	76.56
Dataset	Min	Q1	Median	Q3	Max	Mean
(d) Mistral 7B by datasets.						
Defects4J	0.0	2.25	66.0	93.0	100.0	53.62
SF110	0.0	0.0	2.0	22.0	100.0	15.37
CMD	0.0	0.0	0.0	0.0	0.0	0.0
Prompt	Min	Q1	Median	Q3	Max	Mean
(e) GPT 3.5-turbo on CMD.						
ZSL	2.0	22.5	60.0	91.0	100.0	55.27
FSL	9.0	31.0	81.0	100.0	100.0	67.18
CoT	7.0	93.75	100.0	100.0	100.0	84.65
ToT	28.0	77.25	100.0	100.0	100.0	85.65
GToT	28.0	69.75	100.0	100.0	100.0	84.06
Prompt	Min	Q1	Median	Q3	Max	Mean
(f) GPT 4 on CMD.						
ZSL	0.0	0.0	6.0	61.0	95.0	31.46
FSL	3.0	15.5	51.5	82.25	100.0	51.92
CoT	0.0	18.0	61.0	100.0	100.0	61.05
ToT	0.0	29.0	79.5	100.0	100.0	65.19
GToT	0.0	29.5	79.5	100.0	100.0	65.21

Table 18 Statistical metrics from Method coverage distribution

Prompt	Min	Q1	Median	Q3	Max	Mean
(a) EvoSuite vs GPT 3.5-turbo.						
ZSL	0.0	6.25	83.33	100.0	100.0	56.95
FSL	0.0	7.14	66.67	100.0	100.0	55.31
CoT	0.0	14.29	100.0	100.0	100.0	65.18
ToT	0.0	8.33	66.67	100.0	100.0	55.31
GToT	0.0	16.67	76.92	100.0	100.0	60.32
EvoSuite	0.0	100.0	100.0	100.0	100.0	99.21
Prompt	Min	Q1	Median	Q3	Max	Mean
(b) EvoSuite vs Mistral 7B.						
ZSL	0.0	0.0	29.16	100.0	100.0	48.83
FSL	0.0	9.94	72.22	100.0	100.0	55.1
CoT	0.0	0.0	6.9	97.06	100.0	41.31
ToT	0.0	5.96	50.0	100.0	100.0	55.07
GToT	0.0	17.5	89.59	100.0	100.0	62.56
EvoSuite	0.0	100.0	100.0	100.0	100.0	99.21
Dataset	Min	Q1	Median	Q3	Max	Mean
(c) GPT 3.5-turbo by datasets.						
Defects4J	0.0	33.33	100.0	100.0	100.0	73.86
SF110	0.0	0.0	16.67	89.44	100.0	35.17
CMD	20.0	100.0	100.0	100.0	100.0	89.35
Dataset	Min	Q1	Median	Q3	Max	Mean
(d) Mistral 7B by datasets.						
Defects4J	0.0	17.05	100.0	100.0	100.0	70.1
SF110	0.0	0.0	20.0	50.0	100.0	34.7
CMD	0.0	0.0	0.0	0.0	0.0	0.0
Prompt	Min	Q1	Median	Q3	Max	Mean
(e) GPT 3.5-turbo on CMD.						
ZSL	20.0	35.41	90.91	100.0	100.0	71.29
FSL	25.0	88.89	100.0	100.0	100.0	84.15
CoT	20.0	100.0	100.0	100.0	100.0	91.0
ToT	75.0	100.0	100.0	100.0	100.0	98.04
GToT	75.0	100.0	100.0	100.0	100.0	97.82
Prompt	Min	Q1	Median	Q3	Max	Mean
(f) GPT 4 on CMD.						
ZSL	0.0	0.0	40.0	75.0	100.0	43.56
FSL	16.67	38.33	65.0	100.0	100.0	64.86
CoT	0.0	34.72	100.0	100.0	100.0	69.8
ToT	0.0	62.5	100.0	100.0	100.0	79.59
GToT	0.0	70.46	100.0	100.0	100.0	80.25

Dataset-Level Coverage Trends.

Tables 16, 17, and 18 show coverage varies significantly across datasets. Defects4J achieves highest coverage (GPT-3.5-Turbo: 82.74% line, 85.0% instruction, 100% method), suggesting its structured design facilitates test generation with clearer functional boundaries. In contrast, SF110 reports much lower coverage (3.57% line, 2.0% instruction, 16.67% method) due to higher complexity and diverse architectures. CMD reports highest median coverage (100% for some prompts), though this likely reflects its smaller size and modular structure rather than superior LLM capabilities. The contrast between datasets highlights structural influences: Defects4J benefits from potential training exposure (Sallou et al. (2023)), clearer functional boundaries, and modularized code, while SF110 presents challenges through higher complexity and lower public accessibility.

Finding 21: LLMs achieve higher coverage on well-documented datasets like Defects4J but struggle with complex projects like SF110, where greater structural variability makes test generation more challenging. CMD's high coverage is likely due to its small, modular dataset rather than an inherent LLM advantage.

While Sallou et al. (2023) attributes poor SF110 performance mainly to data accessibility, our analysis demonstrates SF110's inherent complexity complicates test generation (see Table 2 in Section 3.6). However, Defects4J's broad accessibility raises data leakage concerns, as LLMs may have encountered its test cases during pretraining. As shown in Sallou et al. (2023), ChatGPT can retrieve detailed information about specific Defects4J bugs, suggesting performance may reflect memorization rather than genuine reasoning.

Finding 22: Dataset characteristics influence LLM performance—Defects4J's high coverage likely stems from pretraining exposure while SF110's limited representation restricts generalization, highlighting the need for careful dataset selection to mitigate data leakage in test generation studies.

Impact of Reasoning-Based Prompting on Coverage.

Tables 16, 17, 18 show reasoning-based techniques (CoT, GToT, ToT) consistently improve coverage compared to ZSL and FSL.

For line coverage, GPT-3.5-Turbo with CoT achieves highest median (75.47%), while ZSL and FSL produce significantly lower coverage (27.68% and 16.14%). Mistral 7B performs best with GToT (30.0%), though still below GPT-3.5-Turbo.

For instruction coverage, EvoSuite maintains advantage (95.0%). Among LLMs, GPT-3.5-Turbo with CoT (76.0%) and GToT (45.0%) lead, while Mistral 7B's best result with GToT (26.0%) indicates smaller models' limitations.

For method coverage, GPT-3.5-Turbo with CoT reaches 100% in some cases, while Mistral 7B improves with GToT (89.59%) but still lags. These results demonstrate reasoning-based prompting enhances LLM performance significantly but doesn't fully bridge the gap with EvoSuite.

Finding 23: Reasoning-based prompting significantly enhances LLM coverage across all datasets and models. However, smaller models like Mistral 7B exhibit greater limitations even with reasoning-based prompting.

Relation to Prior Work Coverage is central in prior LLM-for-testing evaluations. Siddiq et al. (2024a) report high HumanEval coverage but extremely low SF110 coverage ($< 2\%$), with gains contingent on strong post-fix heuristics. Tang et al. (2024) find ChatGPT achieves $\approx 55\%$ statement coverage on a DynaMOSA-derived benchmark, trailing EvoSuite's $\approx 74\%$ (30 runs). Yang et al. (2024) confirm GPT-4 lags behind EvoSuite on Defects4J, arguing hallucination-driven invalidity is the primary barrier; moreover, directly adapting In-Context Learning (ICL) techniques (CoT/RAG) from other tasks does not improve effectiveness and can reduce it for code-specialized LLMs. Yuan et al. (2024) show iterative compile-feedback (ChatTester) substantially raises compilability/executability, narrowing the gap to manually written tests.

Our results both corroborate and extend these findings. First, evaluating at the class level across Defects4J, SF110, and CMD, we show dataset characteristics strongly shape LLM coverage: relatively higher on Defects4J (potential pretraining exposure), very low on SF110 (structural complexity), and inflated on CMD (small, modular projects). Second, we systematically study prompting strategies and observe reasoning-oriented prompts (e.g., CoT/ToT/GToT) consistently improve coverage—yet still fall short of EvoSuite's systematic exploration. This contrasts with Yang et al. (2024), where CoT and RAG—applied to code-specialized LLMs with a retrieval corpus not tailored to unit tests—are ineffective or detrimental. A plausible explanation: our prompts target developer-facing reasoning on general-purpose LLMs, whereas Yang et al. (2024) adapt In-Context Learning methods from code-generation settings to code-specialized models. Overall, our results indicate coverage improvements from ICL are model- and prompt-design dependent: effective when reasoning aligns with test-generation needs, ineffective when naively ported.

Summary of **RQ5**: EvoSuite consistently outperforms LLMs in structural coverage, achieving near-complete line, instruction, and method coverage. LLMs, however, display higher variability: while Defects4J shows competitive coverage (up to 82% line coverage with GPT-3.5), SF110 highlights their limitations ($< 5\%$) due to architectural complexity. CMD yields artificially high scores, reflecting dataset modularity rather than genuine reasoning ability. Reasoning-based prompting (CoT, ToT, GToT) narrows the gap by substantially improving LLM coverage, but systematic exploration remains a unique strength of SBST approaches like EvoSuite.

4.7 [RQ6]: Test Smell detection.

[Experiment Goal]: Our aim is to evaluate the extent to which the test suites generated by LLMs exhibit test smells compared with those produced by an SBST method like EvoSuite.

[Experiment Design]: This experiment assesses test smell prevalence using TsDetect (Peruma et al. 2020), which identifies 20 different smell types (Peruma et al. 2019, 2020). We record the proportion of test suites containing at least one instance of each smell type. These percentages reflect smell distribution rather than absolute counts, enabling comparative analysis of structural quality and maintainability.

[Experiment Results]:

Test Smell Distribution Across LLMs and EvoSuite.

Table 19 shows that Assertion Roulette (AR), Conditional Logic Test (CLT), and Magic Number Test (MNT) are the most prevalent smells in our setting. The near-ubiquity of MNT across models is consistent with Siddiq et al. (2024a), who analyze multiple smells over two benchmarks (HumanEval and SF110). Our AR variability (e.g., 73.15% with GPT-3.5/FSL on Defects4J vs. 19.86% with GToT on CMD) also falls within the ranges they report. For Exception Handling (EH), we observe higher rates in EvoSuite than in LLM outputs; this differs from patterns noted by Siddiq et al. (2024a) and suggests that reasoning prompts, as part of our prompt engineering (CoT, ToT, GToT), can elicit more explicit exception modeling. We do not observe Default Test, Verbose Test, or Dependent Test in our corpus, consistent with the broader trend that LLM-generated tests tend to be structurally simpler. Across prompt techniques, we see two systematic effects. First, under reasoning prompts (CoT/ToT/GToT), Empty Test (EM) rates increase on CMD and Defects4J, indicating that more structured prompting may improve formatting and organization yet coincide with shallower test bodies (cf. Finding 20). Second, EH tends to increase with reasoning prompts, pointing to more explicit exception scenarios. These patterns persist across model families and benchmarks; notably, MNT remains consistently dominant, with AR and CLT next in prevalence irrespective of the prompt technique.

Finding 24:

Across our benchmarks, LLM-generated tests tend to exhibit simpler structures, showing lower rates of complexity-related smells such as General Fixture (GF: 0.0–35.96%), Redundant Assertion (RA: 0.0–6.17%), and Conditional Logic Test (CLT: 0.0–14.34%) compared to EvoSuite. Conversely, EvoSuite consistently shows higher rates of Exception Handling (EH: 93.10%), Lazy Test (LT: 76.94–82.18%), and Eager Test (EA: 54.67–62.95%) under our configuration, which is consistent with SBST’s emphasis on structural coverage rather than stylistic simplicity.

Dataset-Level Trends in Test Smell Prevalence.

Test smell distribution reveals dataset-specific trends. Defects4J exhibits the highest Assertion Roulette rates (73.15% GPT-3.5-Turbo/FSL), indicating challenges in generating precise assertions for real-world bugs. Our findings partially align with Siddiq et al. (2024a)’s observations, but expand the analysis by comparing SF110, Defects4J, and CMD datasets. SF110 shows elevated General Fixture (93.10% in EvoSuite) and 100% Magic Number Test, while CMD demonstrates the lowest Assertion Roulette (6.4% GPT-3.5-Turbo/CoT). The analysis reveals that LLM-generated tests tend to be stateless and isolated, lacking the inter-method or state-dependent sequences typical of EvoSuite tests. This comprehensive examination illuminates how project context and software evolution influence test smell profiles across different datasets.

Finding 25: LLM-generated CMD tests show limited capturing of shared object states (GF: 0.0–35.96%, MG: 0.0%) compared to EvoSuite (4.44%), revealing challenges in modeling stateful interactions. Defects4J tests display high AR rates (27.97–79.35%), especially in FSL, underlining difficulties in generating precise assertions for real-world defects localization.

Table 19 Test smells distribution by Model, Prompt and by Datasets

Model	Dataset	Prompt	Test Smell (%)																	
			AR	CLT	CI	EM	EH	GF	MG	RP	RA	SE	ST	EA	LT	DA	UT	IT	RO	MNT
GPT 3	SF110	ZSL	46.17	2.68	0.16	12.71	13.02	11.13	2.92	0.0	0.79	23.52	0.39	48.38	53.75	9.31	27.70	0.0	3.24	99.84
		FSL	74.44	3.91	0.11	7.14	14.51	6.70	3.57	0.0	3.13	33.15	0.45	70.65	53.91	16.52	16.29	0.0	4.13	100
		CoT	40.98	2.15	0.0	22.25	11.60	5.93	1.98	0.0	0.95	17.53	0.09	42.44	50.26	6.53	34.54	0.0	1.89	99.05
		ToT	59.77	3.22	0.11	4.94	19.20	13.45	2.41	0.34	3.56	28.51	0.92	56.90	76.21	13.33	15.06	0.11	2.64	100
		GToT	38.23	0.58	0.0	28.97	10.24	2.77	1.99	0.0	2.13	11.11	0.05	42.84	40.03	3.30	37.17	0.0	2.23	100
	Defects4J	ZSL	57.01	4.05	0.06	15.72	24.30	7.50	1.09	0.0	0.91	26.90	0.79	48.55	60.70	9.92	19.65	0.0	1.09	99.64
		FSL	73.15	4.90	0.24	13.18	24.68	3.94	0.88	0.0	2.09	31.19	1.61	64.95	57.64	17.36	15.19	0.0	0.96	100
		CoT	40.06	2.83	0.11	34.40	16.79	2.99	0.61	0.0	0.66	17.73	0.61	36.45	46.81	5.71	41.72	0.0	0.66	99.28
		ToT	69.56	5.52	0.32	4.57	30.60	11.20	1.03	0.0	0.79	29.81	0.79	52.92	75.95	11.20	6.23	0.0	1.0	99.97
		GToT	29.64	2.06	0.0	52.39	12.67	2.03	0.90	0.0	1.11	7.97	0.24	28.42	32.22	1.45	55.05	0.0	1.0	99.97
GPT 4	CMD	ZSL	37.72	0.0	0.0	33.33	15.79	35.96	0.0	0.0	0.0	2.63	0.0	30.70	27.19	0.0	54.39	0.0	0.0	100
		FSL	18.45	0.97	0.0	49.51	25.24	14.56	0.0	0.0	0.0	0.0	0.0	18.45	16.50	0.0	77.67	0.0	0.0	100
		CoT	6.4	0.0	0.0	78.80	7.20	2.80	0.0	0.0	0.40	0.0	0.0	6.40	7.20	0.0	89.20	0.0	0.0	100
		ToT	15.04	0.0	0.0	71.68	7.96	18.58	0.0	0.0	0.0	0.88	0.0	17.70	15.04	0.88	76.99	0.0	0.0	100
		GToT	19.86	0.0	0.0	58.90	15.07	17.81	0.0	0.0	0.68	1.37	0.0	18.49	21.23	0.0	74.66	0.0	0.0	100
	Mistral 7B	ZSL	30.43	1.45	0.0	11.59	14.49	20.29	0.0	0.0	1.45	2.90	0.0	36.23	36.23	0.0	34.78	0.0	0.0	98.55
		FSL	41.18	1.18	0.0	10.59	31.76	20.0	0.0	1.18	0.0	2.35	0.0	48.24	45.88	2.35	36.47	0.0	0.0	100
		CoT	14.56	0.0	0.0	28.16	18.45	17.48	0.0	0.0	0.0	0.97	0.0	18.45	35.92	0.0	54.37	0.0	0.0	98.06
		ToT	16.90	0.0	0.0	33.80	16.90	15.49	0.0	0.0	0.0	0.0	0.0	23.94	22.54	0.0	53.52	0.0	0.0	97.18
		GToT	13.98	0.0	0.0	49.46	17.20	19.35	0.0	0.0	1.08	0.0	0.0	16.13	21.51	0.0	69.89	1.08	0.0	97.85
Defects4J	SF110	ZSL	63.45	9.66	0.0	0.0	14.71	7.98	0.42	1.26	1.26	31.51	0.84	65.13	80.25	10.08	3.36	0.42	0.42	100
		FSL	72.69	11.89	0.0	0.88	7.05	0.44	0.0	1.32	6.17	27.75	0.0	68.28	69.60	15.86	5.29	0.0	0.0	100
		CoT	52.26	7.04	0.0	2.51	19.10	2.01	0.5	1.51	1.51	22.61	0.5	53.27	84.92	11.56	5.53	0.0	0.5	94.47
		ToT	42.59	2.65	0.0	8.47	12.70	2.38	0.26	0.26	1.59	10.58	0.0	43.12	42.59	8.99	11.11	0.0	0.26	92.59
		GToT	21.20	0.82	0.14	13.54	6.57	0.82	0.14	0.0	0.55	3.42	0.0	27.91	16.01	2.74	21.34	0.0	0.14	85.64
	Defects4J	ZSL	73.64	14.34	0.39	0.0	17.44	1.55	0.78	0.78	1.55	13.18	1.55	48.06	58.14	32.17	2.33	0.0	0.39	100
		FSL	79.35	14.17	0.4	1.62	8.5	0.81	0.4	0.0	1.21	25.51	0.81	62.75	78.95	21.46	2.02	0.4	0.4	100

Table 19 (continued)

Model	Dataset	Prompt	Test Smell (%)	AR	CLT	CI	EM	EH	GF	MG	RP	RA	SE	ST	EA	LT	DA	UT	IT	RO	MNT	
		CoT	65.13	13.82	0.0	0.66	21.05	0.66	0.0	0.0	0.0	1.32	10.53	1.32	41.45	81.58	31.58	0.66	0.0	0.0	95.39	
		ToT	69.84	7.54	0.4	1.59	22.62	2.38	0.0	0.0	0.0	1.59	22.22	1.59	64.29	78.97	23.41	3.17	0.0	0.0	0.0	98.41
		GToT	27.97	2.89	0.0	12.86	13.67	0.0	0.0	0.0	0.0	0.16	4.82	0.16	28.94	30.55	4.18	16.08	0.0	0.0	0.0	84.41
CMD		ZSL	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
		FSL	0.0	0.0	0.0	0.0	0.0	100	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	100	0.0	0.0	100
		CoT	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Mixtral8x7B	CMD	ToT	0.0	0.0	0.0	0.0	100	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	100	0.0	0.0	0.0	0.0	
		GToT	16.67	0.0	0.0	33.33	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	16.67	0.0	0.0	33.33	0.0	0.0	66.67
		ZSL	75	0.0	0.0	0.0	25.0	75.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	25.0	25.0	0.0	0.0	0.0	0.0	0.0
EvoSuite	SF110	FSL	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	33.33	0.0	0.0	33.33	0.0	0.0	0.0	66.67
		CoT	0.0	0.0	0.0	0.0	50	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	50.0	0.0	50.0	0.0	0.0	0.0	100
		ToT	44.44	11.11	0.0	44.44	11.11	0.0	0.0	0.0	0.0	0.0	0.0	11.11	0.0	55.56	33.33	0.0	55.56	0.0	0.0	0.0
Defects4J	SF110	GToT	0.0	0.0	0.0	50.0	50.0	50.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	50.0	0.0	100	0.0	0.0	100
		FSL	50.78	0.0	0.0	6.90	93.10	0.0	4.44	0.0	0.0	0.0	17.53	0.0	54.67	76.94	0.07	6.90	0.0	3.96	0.0	100
		ToT	38.24	0.0	0.0	6.90	93.10	0.0	4.44	0.0	0.0	0.0	10.71	0.0	62.95	82.18	6.73	0.38	0.0	4.34	0.0	100

* AR: Assertion Roulette, CLT: Conditional Logic Test, CI: Constructor Initialization, EM: Empty Test, EH: Exception Handling, GF: General Fixture, MG: Mystery Guest, RP: Redundant Print, RA: Redundant Assertion, SE: Sensitive Equality, ST: Sleepy Test, EA: Eager Test, LT: Lazy Test, DA: Duplicate Assert, UT: Unknown Test, IT: Ignored Test, RO: Resource Optimism, MNT: Magic Number Test, ZSL: Zero-shot Learning, FSL: Few-shot Learning, CoT: Chain-of-Thought, ToT: Tree-of-Thoughts, GToT: Guided Tree-of-Thoughts

Impact of Prompt Engineering on Test Quality.

Test smell prevalence varies across LLMs, with CoT and GToT prompts improving quality. GPT-4 (CMD) reduces Assertion Roulette to 14.56%, and GPT-3.5-Turbo achieves the lowest AR at 6.4% (CoT on CMD), yet both still show nearly 100% Magic Number Test. Mistral 7B and Mixtral 8x7B yield more smells, with Mistral 7B reaching 27.97% AR on Defects4J. EvoSuite, though competitive, exhibits 100% MNT and notable AR rates (38.24–50.78%). These findings align with Siddiq et al. (2024a)'s report of universally high MNT and variable AR. Our results extend their analysis by evaluating more models and prompts, demonstrating how reasoning-based prompting reduces AR. This prompt-model-smell interplay emphasizes that improvements depend on structured prompting strategies, not just model capabilities. MNT's persistence confirms its status as a core limitation in current test generation methods.

Finding 26: CoT and GToT produce the cleanest tests, notably reducing AR smells (GPT-3 CoT: 6.4–40.98%, GPT-3 GToT: 19.86–38.23%, GPT-4 CoT: 14.56%) compared to FSL (18.45–79.35%) and EvoSuite (38.24–50.78%). However, all strategies consistently struggle with Magic Number smells (85.64–100%), highlighting ongoing challenges with numeric literals.

Relation to Prior Work Siddiq et al. (2024a) present the first systematic analysis of test smells in LLM-generated tests, applying TsDetect on HumanEval and SF110. They report near-universal Magic Number (MNT) and Lazy Test (LT) smells, with Assertion Roulette (AR) and Eager Test (EA) also prevalent, while EvoSuite exhibited higher rates of Exception Handling (EH) and LT. Their study focuses on three models (Codex, GPT-3.5, StarCoder) and two benchmarks, providing an important initial view of how smells manifest in generated tests. Our work extends this line by covering a broader spectrum of LLM families and prompting strategies across three benchmarks (Defects4J, SF110, CMD), and by contrasting them against EvoSuite. We confirm the persistence of MNT, but show that reasoning-based prompting substantially reduces AR while EvoSuite continues to exhibit higher EH, LT, and EA, highlighting complementary strengths and weaknesses between SBST and LLM-based approaches.

Summary of **RQ6:** LLM-generated tests show distinct smell profiles compared to EvoSuite. Magic Number Tests remain pervasive (85–100% across all settings), but reasoning-based prompts (CoT, GToT) cut Assertion Roulette rates to as low as 6.4% (vs. up to 79.3% with FSL and 38–51% with EvoSuite). EvoSuite, in turn, exhibits high Lazy Test (77–82%) and Exception Handling (93%) smells, underscoring its coverage-oriented bias. Overall, LLMs yield structurally simpler tests with fewer stateful smells (e.g., GF \geq 36% vs. 93% in EvoSuite), and prompt design emerges as a decisive factor in mitigating assertion-related issues. These results extend Siddiq et al. (2024a) by quantifying prompt-driven shifts in smell prevalence across multiple datasets and models.

5 Discussion

This section discusses the potential threats to validity, limitations of our study, and suggestions for future work.

5.1 Threats to Validity

This section addresses the threats to the validity of this study.

External Validity.

The generalizability of our findings faces several limitations. Though we used three datasets (Defects4J, SF110, and CMD), these may not fully represent real-world software complexity. CMD's small size of just two projects restricts our observations, particularly regarding the unexpected cases where GPT-3.5-Turbo outperformed GPT-4, suggesting model performance varies significantly with dataset characteristics and prompting approaches. Further experimentation on a broader dataset is necessary to confirm our observations.

Despite our efforts to mitigate training data contamination through CMD, LLMs may still have encountered similar test patterns during pretraining, potentially overestimating their true capabilities—a persistent challenge in LLM-based software engineering research. Our evaluation of four models from OpenAI and MistralAI further limits the breadth of our conclusions. Future work should explore a wider range of models, including open-source LLMs trained specifically for software engineering tasks.

Model knowledge cutoff dates also likely influenced our results, with Mistral 7B's older knowledge base (2021-2022) compared to Mixtral 8x7B's more recent release (December 2023) creating performance disparities. Our findings reflect these models' capabilities during our study period (November 2023–April 2024), and newer versions may demonstrate different results.

Internal Validity.

Several internal threats could affect our results' reliability. LLM output randomness introduces variability in generated tests, which we mitigated by executing each prompt 30 times per test suite and performing statistical analyses. Similarly, we ran 30 iterations to address EvoSuite's variability from its evolutionary search process.

Test smell detection methodology presents another challenge. Recent studies (Panichella et al. 2022) highlight concerns about detection tools' reliability, noting high false-positive and false-negative rates. Some smells like Assertion Roulette and Eager Test don't always indicate poor maintainability in LLM-generated tests. To mitigate this, we manually validated a subset of test cases to better assess whether detected smells genuinely impact maintainability, though this doesn't eliminate all possible biases.

Finally, prompting strategy effectiveness represents another concern. While GToT consistently improves test structure and reliability, prompt designs may need further refinement for different models and datasets. Future research should explore model-specific prompt tuning to optimize LLM-generated test quality.

5.2 Limitations and Future Work

5.2.1 Limitations.

Our study does not assess fault detection. Specifically, we did not implement (i) regression testing on paired buggy and fixed versions (i.e., generating tests on the fixed version and executing them on the buggy counterpart, as in Shamshiri et al. (2015)), (ii) the exception-oriented fault-detection setup used in recent LLM-vs-SBST comparisons (e.g., Tang et al. (2024)), which was introduced to sidestep unreliable automatically generated assertions but inherently captures only faults that manifest as runtime exceptions, or (iii) mutation-based analyses. Extending our evaluation to fault detection requires dependable oracles at scale and datasets instrumented for paired versions or mutation loops, which are outside the scope of the present study and outlined in Future Work. We also limited our comparison to EvoSuite as a single SBST baseline to maintain a clear experimental scope focused on LLM-generated test suites. Other automated test generation tools such as fuzzers, random testing, or symbolic execution frameworks were not included. We acknowledge the value of these alternatives and consider their integration an important direction for future work.

5.2.2 Future Work.

Regression testing on buggy/fixed pairs. We will re-instrument our benchmarks to enable a principled regression protocol (generate tests on the fixed version and execute them on the buggy counterpart) so that fault-revealing capability can be assessed with a controlled oracle, following established practice on Defects4J.

Principled, mutation-guided pipelines. Rather than treating mutation score as a raw end-metric, we will adopt mutation-guided loops in which surviving mutants drive subsequent test improvements (prompt augmentation/repair), with pre-filters ensuring tests are executable and assertive before mutation. This direction is supported by MuTAP (Dakhel et al. 2024), which demonstrates that leveraging mutants as feedback can substantially raise fault-revealing effectiveness.

Hybrid SBST + LLM with full-fledged refactoring. We will combine SBST (for structural coverage and input diversity) with LLM-based refactoring beyond identifier naming (e.g., clarifying fixtures and test scaffolding, restructuring helpers, strengthening oracles, reducing brittle mocks or magic values, and improving assertion intent) so that readability and maintainability gains translate into developer effectiveness. Deljouyi et al. (2024) already show that LLM-enhanced SBST tests can improve bug-fix performance (up to +33% bugs fixed, -20% time) and that semantics-preserving renaming can make generated tests as readable as developer-written tests. Our next step is to generalize these benefits to structural refactoring of test code (layout, fixtures, oracles), not only surface-level names or comments.

Controlled oracles at scale. Finally, we will explore non-LLM-derived oracles (e.g., property- or metamorphic-style checks, or contract-based assertions) to reduce ambiguity in execution outcomes and make exception-only detection unnecessary, addressing a key limitation observed in recent LLM-vs-SBST comparisons.

Extended baselines. We also plan to broaden our evaluation by incorporating other automated test generation tools such as fuzzers, random test generators, and symbolic/concolic engines. This will enable a more comprehensive assessment of LLM-based test generation in relation to diverse testing paradigms.

6 Related work

6.1 LLM-Based Test Generation: Strengths and Limitations

Recent studies have assessed LLMs for unit test generation along multiple dimensions, not limited to static properties. For instance, Siddiq et al. (2024a) evaluate syntactic and compilation correctness, coverage, and the presence of test smells across HumanEval and SF110, showing that while coverage can be competitive in some cases, validity and smell-related issues remain frequent. Yuan et al. (2024) conduct both a quantitative analysis and a user study, reporting that although many LLM-generated tests fail to compile or execute, the passing ones can achieve coverage and readability close to human-written tests; moreover, iterative refinement (ChatTester) improves compilability and assertion correctness. Comparative works have also positioned LLMs against SBST tools: Tang et al. (2024) examine correctness, readability, coverage, and bug detection versus EvoSuite; Yang et al. (2024) provide a broad empirical evaluation across different LLMs and datasets, concluding that LLMs still underperform EvoSuite in coverage, largely due to validity and dependency resolution issues.

Overall, prior work has already examined runtime behavior (e.g., execution and coverage) and aspects of maintainability (e.g., readability, smells). Our study complements this body of evidence by broadening scale and scope, covering three datasets with class-level generation, multiple LLM families, and five prompting strategies and by structuring failures into a fine-grained taxonomy that highlights hallucination-driven issues. We explicitly avoid exclusive claims and instead position our results as an extension and consolidation across settings and prompts.

6.2 The Impact of Prompt Engineering in Test Generation

The impact of prompt engineering on LLM-based test generation has been explored in recent research. Yuan et al. (2024) and Chen et al. (2024) examined the effects of different prompting techniques on test correctness, demonstrating that Few-Shot Learning (FSL) improves correctness but does not fully mitigate hallucinated dependencies. Similarly, Yang et al. (2024) introduced Chain-of-Thought (CoT) prompting, highlighting its ability to enhance logical reasoning in test generation.

Building upon these insights, our study systematically compares five prompting strategies namely ZSL, FSL, CoT, ToT, and GToT across multiple LLMs and datasets, focusing on structural correctness, compilability, coverage, and maintainability to assess their practical effectiveness in automated unit test generation.

6.3 Readability, Maintainability, and Test Smell Analysis

Aspects of maintainability have been discussed in prior work, though often in limited scope. Siddiq et al. (2024a) quantify the prevalence of smells such as duplicated asserts and empty tests in LLM-generated suites, reporting nontrivial rates particularly on SF110. Yuan et al. (2024) complement this with a user study indicating that passing LLM-generated tests can achieve readability close to human-written ones, while also noting frequent failures and incorrect assertions in the broader population. Beyond Java, Schäfer et al. (2023) show that LLM-generated JavaScript tests can include meaningful oracles and competitive coverage, though such results depend strongly on runtime and language-specific contexts.

Building on these observations, our study provides a broader and more systematic assessment of maintainability. We analyze test smells using TsDetect, evaluate readability with a specialized deep learning model for JUnit tests (Scalabrino et al. 2018), and incorporate structural measures such as Match Success Rate (MSR) and Code Extraction Success Rate (CSR). In contrast to prior evaluations, we scale this analysis to three datasets with class-level test generation, multiple LLM families, and five prompting strategies, explicitly examining how reasoning-based prompting modulates maintainability indicators. This comprehensive approach uncovers structural and readability challenges in LLM-generated tests and provides actionable insights for prompt design and post-processing aimed at improving long-term usability.

6.4 Advancing LLM-Based Unit Test Improvement

Recent studies have explored LLM-driven unit test improvement techniques, primarily focusing on iterative refinement. Gu et al. (2024) introduced TestArt, a co-evolutionary approach that iteratively refines LLM-generated tests through repair iterations, enhancing correctness over multiple steps. Similarly, Zhang et al. (2025) proposed CITYWALK, which employs iterative refinement guided by contextual cues to improve generated tests, with a focus on increasing functional correctness. Alshahwan et al. (2024) investigated LLM-driven test improvements at Meta, focusing on refining correctness but without addressing key aspects such as maintainability, readability, or test smells. These studies demonstrate progress in test improvement but remain limited to correctness-focused refinements.

In contrast, our work provides a multi-dimensional assessment that goes beyond correctness-focused refinements. We complement prior improvement techniques by evaluating structure- and maintenance-oriented qualities: extractability (MSR/CSR), syntactic correctness, compilability, coverage, readability, and test smells. Moreover, we introduce readability and maintainability metrics to assess whether generated tests are understandable and practically usable in real-world software projects.

Our findings indicate that while iterative refinement can enhance the syntactic and structural quality of LLM-generated tests, it does not fully address maintainability and structural robustness: compilability remains fragile and test smells persist across datasets and prompting strategies. This suggests the need for hybrid approaches that combine Search-Based Software Testing (SBST) with LLM-based generation to jointly target correctness and maintainability.

6.5 Key Takeaways and Research Gap

Prior works on LLM-based test generation primarily focus on syntactic correctness and code coverage, leaving key aspects of readability, maintainability, and structural reliability underexplored. Our study fills this gap by providing a comprehensive evaluation framework that goes beyond static correctness checks to assess multiple dimensions of test quality. Specifically, we evaluate test readability and maintainability using deep learning-based readability scoring, aligned with human evaluation guidelines.

Unlike previous studies that explore only one or two prompting techniques, we systematically compare five strategies—ZSL, FSL, CoT, ToT, and GToT. We also integrate mutation testing, a missing component in prior research, to evaluate fault detection capabilities. Beyond correctness and structural coverage (line, instruction, method), we analyze test smells, code extraction reliability (MSR & CSR), compilation errors, and runtime failures, uncovering key challenges in dependency resolution and API usage. Unlike previous studies that explore only one or two prompting techniques, we systematically compare five strategies (ZSL, FSL, CoT, ToT, and GToT) across four LLMs and three datasets. Beyond structural correctness and coverage (line, instruction, and method), we analyze code extractability (MSR & CSR), compilation success, and test smells to uncover recurring issues in dependency resolution and API usage.

A key differentiator of our work is the diversity of datasets: SF110 (academic), Defects4J (real-world bugs), and CMD (industrial codebase). This design ensures that our findings generalize across heterogeneous software environments. By bridging structural reliability, readability, maintainability, and dataset diversity, our study establishes a broad, data-driven foundation for understanding the practical effectiveness and current limitations of LLM-generated test suites.

7 Discussion of Findings and Implications

Our study reveals that reasoning-based prompting (CoT, GToT) improves the structural correctness and maintainability of LLM-generated tests, yet reliability and integration robustness remain major challenges. While many tests compile successfully, they often exhibit dependency mismatches or redundant logic, and test smells continue to affect maintainability. Hybrid strategies combining Search-Based Software Testing (SBST) with LLM-driven generation and targeted human oversight appear essential for achieving higher robustness. Although prompt engineering enhances structural reliability, hallucination-induced errors persist, highlighting the need for stronger contextual grounding and improved handling of dependencies and encapsulation. These findings underscore the importance of adaptive test generation pipelines that integrate automated validation, dependency resolution, and iterative refinement to improve the practical usability of LLM-generated unit tests.

7.1 Implications for Researchers

Our findings emphasize reasoning-based prompting's significant role in improving LLM-generated test suites. Unlike prior studies (Siddiq et al. 2024a; Tang et al. 2024; Yang et al. 2024), which did not extensively analyze prompt engineering, we systematically evaluate

ZSL, FSL, CoT, ToT, and GToT. Results show that structured reasoning significantly enhances test adherence, compilability, and structural reliability, with GToT consistently outperforming other strategies.

Our multi-level coverage analysis reveals that LLM-generated tests achieve moderate coverage but remain inferior to EvoSuite, reinforcing the need for hybrid approaches that integrate SBST techniques with LLM-based generation. Our findings also challenge the common assumption that larger models inherently produce better tests: GPT-3.5-turbo occasionally outperforms GPT-4 on CMD, highlighting the need to better understand how architectural and dataset characteristics influence performance.

Beyond correctness and coverage, our readability and maintainability analysis reveals that while LLMs produce syntactically correct and readable tests, readability alone does not guarantee maintainability. Test smells—particularly Magic Number Tests and Assertion Roulette—persist across models and prompting strategies, increasing debugging complexity. These results highlight the need for more refined prompting strategies and automated post-processing techniques to reduce test smells and improve long-term test reliability.

7.2 Implications for Developers

Our study highlights the practical value of LLM-generated test suites when readability and maintainability are prioritized over raw coverage. Unlike EvoSuite, which maximizes coverage at the expense of readability, LLM-generated tests generally exhibit clearer structure, more meaningful naming, and higher comprehensibility—particularly when reasoning-based prompting techniques such as GToT and CoT are applied. This makes these tests more accessible for long-term maintenance and collaborative development.

However, developers must remain aware of reliability concerns. While LLMs can generate large numbers of tests rapidly, issues such as partial compilability, redundant assertions, or unverified logic can still arise. Persistent test smells—including redundant or fragile assertions, flaky behavior, and inconsistent naming—emphasize the importance of human oversight. LLMs thus serve best as “assistive tools” rather than fully autonomous test generators. Careful prompt refinement, automated filtering, and lightweight post-processing validation can maximize their practical utility, ensuring that generated tests meaningfully contribute to software quality and long-term maintainability.

8 Conclusion

This study presents a large-scale evaluation of Large Language Models (LLMs) for automated unit test generation, systematically analyzing how structured prompting strategies influence test quality, readability, and maintainability across multiple datasets. By comparing five prompting techniques (ZSL, FSL, CoT, ToT, and GToT) and four LLMs, our results show that reasoning-based prompting especially GToT and CoT substantially improves test structure, compilability, and readability compared to zero-shot prompting. However, LLM-generated tests remain limited in robustness and reliability. Although they exhibit cleaner code and higher readability than traditional SBST-generated tests, persistent issues such as incomplete imports, redundant assertions, and recurring test smells (e.g., Magic Number Tests, Assertion Roulette) reduce their long-term maintainability.

Coverage analysis further reveals that while LLMs achieve moderate coverage, they still underperform EvoSuite in exploring program structure. Overall, our findings indicate that LLMs can indeed assist in automated test generation, but they are not yet ready to replace traditional testing tools. Their greatest strength lies in producing human-readable, maintainable test scaffolds that can complement existing SBST or developer-written tests. To unlock their full potential, future research should focus on hybrid methodologies that integrate LLM-based generation with automated validation, dependency resolution, and search-based refinement. Such approaches could enable LLMs to evolve from assistive code generators into reliable components of software quality assurance pipelines.

Author Contributions All authors contributed equally to this work. They jointly conceived the study, conducted the experiments, analyzed the results, and wrote the manuscript. All authors reviewed and approved the final version of the manuscript.

Funding This research was funded in whole, or in part, by the Luxembourg National Research Fund (FNR), grant reference AFR PhD bilateral, project reference 17185670.

This work was also supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 949014).

For the purpose of open access, and in fulfilment of the obligations arising from the grant agreement, the author has applied a Creative Commons Attribution 4.0 International (CC BY 4.0) license to any Author Accepted Manuscript version arising from this submission.

Data Availability The datasets and code used in the present study are available in our repository: <https://anonymous.4open.science/r/LLM4TS-0F76/>

Declarations

Ethical Approval This article does not contain any studies with human participants or animals performed by any of the authors.

Conflicts of Interest The authors declare that they have no conflict of interest.

Informed Consent Not applicable.

Clinical Trial Number Not applicable.

References

- Abdullin AM, Itsykson VM (2022) Kex: A platform for analysis of jvm programs. *Inform Contr Syst* 1(116):30–43
- Achiam J, Adler S, Agarwal S, Ahmad L, Akkaya I, Aleman FL, Almeida D, Altschmidt J, Altman S, Anadkat S et al (2023) Gpt-4 technical report. [arXiv:2303.08774](https://arxiv.org/abs/2303.08774)
- Almasi MM, Hemmati H, Fraser G, Arcuri A, Benefelds J (2017) An industrial evaluation of unit test generation: Finding real faults in a financial application. In: 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), IEEE, pp 263–272
- Alshahwan N, Chheda J, Finogenova A, Gokkaya B, Harman M, Harper I, Marginean A, Sengupta S, Wang E (2024) Automated unit test improvement using large language models at meta. In: Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, pp 185–196
- Amatriain X (2024) Prompt design and engineering: Introduction and advanced methods. *arXiv preprint [arXiv:2401.14423](https://arxiv.org/abs/2401.14423)*
- Arcuri A, Fraser G (2013) Parameter tuning or default values? an empirical investigation in search-based software engineering. *Empir Softw Eng* 18:594–623

- Arcuri A, Fraser G, Just R (2017) Private api access and functional mocking in automated unit test generation. In: 2017 IEEE international conference on software testing, verification and validation (ICST), IEEE, pp 126–137
- Beck K (2000) Extreme programming explained: embrace change. Addison-Wesley Professional
- Biagiola M, Ghisloti G, Tonella P (2025) Improving the readability of automatically generated tests using large language models. 2025 IEEE Conference on Software Testing, Verification and Validation (ICST), IEEE, pp 162–173
- Brown T, Mann B, Ryder N, Subbiah M, Kaplan JD, Dhariwal P, Neelakantan A, Shyam P, Sastry G, Askell A et al (2020) Language models are few-shot learners. *Adv Neural Inf Process Syst* 33:1877–1901
- Buse RP, Weimer WR (2009) Learning a metric for code readability. *IEEE Trans Software Eng* 36(4):546–558
- Chen Y, Hu Z, Zhi C, Han J, Deng S, Yin J (2023) Chatunitest: A framework for llm-based test generation. *arXiv e-prints* pp arXiv-2305
- Chen Y, Hu Z, Zhi C, Han J, Deng S, 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, pp 572–576
- Daka E, Fraser G (2014) A survey on unit testing practices and problems. In: 2014 IEEE 25th International Symposium on Software Reliability Engineering, IEEE, pp 201–211
- Daka E, Campos J, Fraser G, Dorn J, Weimer W (2015) Modeling readability to improve unit tests. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, pp 107–118
- Dakheel AM, Nikanjam A, Majdinasab V, Khomh F, Desmarais MC (2024) Effective test generation using pre-trained large language models and mutation testing. *Inf Softw Technol* 171:107468
- Dantas CEC, Maia MA (2021) Readability and understandability scores for snippet assessment: An exploratory study. *arXiv preprint arXiv:2108.09181*
- Deljouyi A, Koohestani R, Izadi M, Zaidman A (2024) Leveraging large language models for enhancing the understandability of generated unit tests. *arXiv preprint arXiv:2408.11710*
- Deljouyi A, Koohestani R, Izadi M, Zaidman A (2025) Leveraging large language models for enhancing the understandability of generated unit tests. In: 2025 IEEE/ACM 47th international conference on software engineering (icse). Los Alamitos, CA, USA pp 392–404
- Dorn J (2012) A general software readability model. MCS Thesis available from (<http://www.cs.virginia.edu/~weimer/students/dorn-mcs-paper.pdf>) 5:11–14
- Fan A, Gokkaya B, Harman M, Lyubarskiy M, Sengupta S, Yoo S, Zhang JM (2023) Large language models for software engineering: Survey and open problems. *arXiv preprint arXiv:2310.03533*
- Fraser G, Arcuri A (2011) Evosuite: automatic test suite generation for object-oriented software. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, pp 416–419
- Fraser G, Arcuri A (2013) Evosuite: On the challenges of test case generation in the real world. In: 2013 IEEE sixth international conference on software testing, verification and validation, IEEE, pp 362–369
- Fraser G, Staats M, McMinn P, Arcuri A, Padberg F (2015) Does automated unit test generation really help software testers? a controlled empirical study. *ACM Trans Softw Eng Methodol (TOSEM)* 24(4):1–49
- Grano G, Scalabrino S, Gall HC, Oliveto R (2018) An empirical investigation on the readability of manual and generated test cases. In: Proceedings of the 26th Conference on Program Comprehension, pp 348–351
- Gu S, Fang C, Zhang Q, Tian F, Chen Z (2024) Testart: Improving llm-based unit test via co-evolution of automated generation and repair iteration. *arXiv e-prints* pp arXiv-2408
- Hossain SB, Dwyer M (2024) Togl: Correct and strong test oracle generation with llms. *arXiv preprint arXiv:2405.03786*
- Jahangirova G, Terragni V (2023) Sbft tool competition 2023-java test case generation track. In: 2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT), IEEE, pp 61–64
- Jiang AQ, Sablayrolles A, Mensch A, Bamford C, Chaplot DS, Casas Ddl, Bressand F, Lengyel G, Lample G, Saulnier L et al (2023) Mistral 7b. *arXiv preprint arXiv:2310.06825*
- Jiang AQ, Sablayrolles A, Roux A, Mensch A, Savary B, Bamford C, Chaplot DS, Casas Ddl, Hanna EB, Bressand F et al (2024) Mixtral of experts. *arXiv preprint arXiv:2401.04088*
- Kojima T, Gu SS, Reid M, Matsuo Y, Iwasawa Y (2022) Large language models are zero-shot reasoners. *Adv Neural Inf Process Syst* 35:22199–22213
- Kumar A, Haiduc S, Das PP, Chakrabarti PP (2024) Llms as evaluators: A novel approach to evaluate bug report summarization. *arXiv preprint arXiv:2409.00630*
- Li J, Li G, Li Y, Jin Z (2025) Structured chain-of-thought prompting for code generation. *ACM Trans Softw Eng Methodol* 34(2):1–23
- Long J (2023) Large language model guided tree-of-thought. *arXiv preprint arXiv:2305.08291*
- Macedo M, Tian Y, Cogo FR, Adams B (2024) Exploring the impact of the output format on the evaluation of large language models for code translation. *arXiv preprint arXiv:2403.17214*


- Mi Q, Hao Y, Ou L, Ma W (2022) Towards using visual, semantic and structural features to improve code readability classification. *J Syst Softw* 193:111454
- Molina F, Gorla A, d'Amorim M (2025) Test oracle automation in the era of llms. *ACM Trans Softw Eng Methodol* 34(5):1–24
- Naveed H, Khan AU, Qiu S, Saqib M, Anwar S, Usman M, Barnes N, Mian A (2023) A comprehensive overview of large language models. arXiv preprint [arXiv:2307.06435](https://arxiv.org/abs/2307.06435)
- Oliveira D, Bruno R, Madeiral F, Masuhara H, Castor F (2022) A systematic literature review on the impact of formatting elements on program understandability. Available at SSRN 4182156
- OpenAI (2023) Gpt-3.5-turbo. <https://platform.openai.com/docs/models/gpt-3-5-turbo/>. Accessed 21 May 2024
- Ouédraogo WC, Plein L, Kabore K, Habib A, Klein J, Lo D, Bissyandé TF (2025) Enriching automatic test case generation by extracting relevant test inputs from bug reports. *Empir Softw Eng* 30(3):85
- Pacheco C, Ernst MD (2007) Randoop: feedback-directed random testing for java. In: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion, pp 815–816
- Palomba F, Di Nucci D, Panichella A, Oliveto R, De Lucia A (2016) On the diffusion of test smells in automatically generated test code: An empirical study. In: Proceedings of the 9th international workshop on search-based software testing, pp 5–14
- Panichella A, Kifetew FM, Tonella P (2017) Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Trans Software Eng* 44(2):122–158
- Panichella A, Panichella S, Fraser G, Sawant AA, Hellendoorn VJ (2020) Revisiting test smells in automatically generated tests: limitations, pitfalls, and opportunities. In: 2020 IEEE international conference on software maintenance and evolution (ICSME), IEEE, pp 523–533
- Panichella A, Panichella S, Fraser G, Sawant AA, Hellendoorn VJ (2022) Test smells 20 years later: detectability, validity, and reliability. *Empir Softw Eng* 27(7):170
- Peruma A, Almalki KS, Newman CD, Mkaouer MW, Ouni A, Palomba F (2019) On the distribution of test smells in open source android applications: An exploratory study. (2019). Citado na p 13
- Peruma A, Almalki K, Newman CD, Mkaouer MW, Ouni A, Palomba F (2020) Tsdetect: An open source test smells detection tool. In: Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering, pp 1650–1654
- Pinto GH, Vergilio SR (2010) A multi-objective genetic algorithm to test data generation. In: 2010 22nd IEEE International Conference on Tools with Artificial Intelligence, IEEE, vol 1, pp 129–134
- Posnett D, Hindle A, Devanbu P (2011) A simpler model of software readability. In: Proceedings of the 8th working conference on mining software repositories, pp 73–82
- Rojas JM, Fraser G, Arcuri A (2015) Automated unit test generation during software development: A controlled experiment and think-aloud observations. In: Proceedings of the 2015 international symposium on software testing and analysis, pp 338–349
- Sahoo P, Singh AK, Saha S, Jain V, Mondal S, Chadha A (2024) A systematic survey of prompt engineering in large language models: Techniques and applications. arXiv preprint [arXiv:2402.07927](https://arxiv.org/abs/2402.07927)
- Sallou J, Durieux T, Panichella A (2023) Breaking the silence: the threats of using llms in software engineering. arXiv preprint [arXiv:2312.08055](https://arxiv.org/abs/2312.08055)
- Scalabrino S, Linares-Vásquez M, Oliveto R, Poshyvanyk D (2018) A comprehensive model for code readability. *J Softw Evol Process* 30(6):e1958
- Schäfer M, Nadi S, Eghbali A, Tip F (2023) An empirical evaluation of using large language models for automated unit test generation. *IEEE Trans Softw Eng*
- Sen K, Marinov D, Agha G (2005) Cute: A concolic unit testing engine for c. *ACM SIGSOFT Softw Eng Notes* 30(5):263–272
- Sergeyuk A, Lvova O, Titov S, Serova A, Bagirov F, Kirillova E, Bryksin T (2024) Reassessing java code readability models with a human-centered approach. In: Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension, pp 225–235
- Shamshiri S, Just R, Rojas JM, Fraser G, McMinn P, Arcuri A (2015) Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, pp 201–211
- Shamshiri S, Rojas JM, Galeotti JP, Walkinshaw N, Fraser G (2018) How do automatically generated unit tests influence software maintenance? In: 2018 IEEE 11th international conference on software testing, verification and validation (ICST), IEEE, pp 250–261
- Si C, Gan Z, Yang Z, Wang S, Wang J, Boyd-Graber J, Wang L (2022) Prompting gpt-3 to be reliable. arXiv preprint [arXiv:2210.09150](https://arxiv.org/abs/2210.09150)

- Siddiq ML, Da Silva Santos JC, Tanvir RH, Ulfat N, Al Rifat F, Carvalho Lopes V (2024a) Using large language models to generate junit tests: An empirical study. In: Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering, pp 313–322
- Siddiq ML, Dristi S, Saha J, Santos J (2024b) Quality assessment of prompts used in code generation. arXiv preprint [arXiv:2404.10155](https://arxiv.org/abs/2404.10155)
- Sun W, Miao Y, Li Y, Zhang H, Fang C, Liu Y, Deng G, Liu Y, Chen Z (2024) Source code summarization in the era of large language models. arXiv preprint [arXiv:2407.07959](https://arxiv.org/abs/2407.07959)
- Tang Y, Liu Z, Zhou Z, Luo X (2024) Chatgpt vs sbst: A comparative assessment of unit test suite generation. *IEEE Trans Softw Eng*
- Wang J, Huang Y, Chen C, Liu Z, Wang S, Wang Q (2024a) Software testing with large language models: Survey, landscape, and vision. *IEEE Trans Softw Eng*
- Wang T, Zhou N, Chen Z (2024b) Enhancing computer programming education with llms: A study on effective prompt engineering for python code generation. arXiv preprint [arXiv:2407.05437](https://arxiv.org/abs/2407.05437)
- Wang X, Wei J, Schuurmans D, Le Q, Chi E, Narang S, Chowdhery A, Zhou D (2022) Self-consistency improves chain of thought reasoning in language models. arXiv preprint [arXiv:2203.11171](https://arxiv.org/abs/2203.11171)
- Wei J, Wang X, Schuurmans D, Bosma M, Xia F, Chi E, Le QV, Zhou D et al (2022) Chain-of-thought prompting elicits reasoning in large language models. *Adv Neural Inf Process Syst* 35:24824–24837
- Winkler D, Urbanke P, Ramler R (2024) Investigating the readability of test code. *Empir Softw Eng* 29(2):53
- Yang L, Yang C, Gao S, Wang W, Wang B, Zhu Q, Chu X, Zhou J, Liang G, Wang Q et al (2024) On the evaluation of large language models in unit test generation. In: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, pp 1607–1619
- Yuan Z, Lou Y, Liu M, Ding S, Wang K, Chen Y, Peng X (2023) No more manual tests? evaluating and improving chatgpt for unit test generation. arXiv preprint [arXiv:2305.04207](https://arxiv.org/abs/2305.04207)
- Yuan Z, Liu M, Ding S, Wang K, Chen Y, Peng X, Lou Y (2024) Evaluating and improving chatgpt for unit test generation. *Proceed ACM Softw Eng* 1(FSE):1703–1726
- Zeller A, Gopinath R, Böhme M, Fraser G, Holler C (2019) The fuzzing book
- Zhang X, Hou X, Qiao X, Song W (2024a) A review of automatic source code summarization. *Empir Softw Eng* 29(6):162
- Zhang Y, Lu Q, Liu K, Dou W, Zhu J, Qian L, Zhang C, Lin Z, Wei J (2025) Citywalk: Enhancing llm-based c++ unit test generation via project-dependency awareness and language-specific knowledge. arXiv preprint [arXiv:2501.16155](https://arxiv.org/abs/2501.16155)
- Zhang Z, Wang Y, Wang C, Chen J, Zheng Z (2024b) Llm hallucinations in practical code generation: Phenomena, mechanism, and mitigation. arXiv preprint [arXiv:2409.20550](https://arxiv.org/abs/2409.20550)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.

Authors and Affiliations

Wendkûuni C. Ouédraogo¹ · Abdoul Kader Kaboré¹ · Yinghua Li²  · Haoye Tian⁴ · Anil Koyuncu⁵ · Jacques Klein¹ · David Lo³ · Tegawendé F. Bissyandé¹

✉ Yinghua Li
yinghua.li@njust.edu.cn

Wendkûuni C. Ouédraogo
wendkuuni.ouedraogo@uni.lu

Abdoul Kader Kaboré
abdoulkader.kabore@uni.lu

Haoye Tian
haoye.tian@aalto.fi

Anil Koyuncu
anil.koyuncu@cs.bilkent.edu.tr

Jacques Klein
jacques.klein@uni.lu

David Lo
davidlo@smu.edu.sg

Tegawendé F. Bissyandé
tegawende.bissyande@uni.lu

¹ SnT Centre, University of Luxembourg, Esch-sur-Alzette, Luxembourg

² Nanjing University of Science and Technology, Nanjing, China

³ Singapore Management University, Singapore, Singapore

⁴ Aalto University, Espoo, Finland

⁵ Bilkent University, Ankara, Turkey