

Seed&Steer: Guiding Large Language Models with Compilable Prefix and Branch Signals for Unit Test Generation

Shuaiyu Zhou
Peking University
Beijing, China
monster290@stu.pku.edu.cn

Zhengran Zeng
Peking University
Beijing, China
zhengranzeng@stu.pku.edu.cn

Xiaoling Zhou
Peking University
Beijing, China
xiaolingzhou@stu.pku.edu.cn

Rui Xie
Peking University
Beijing, China
ruixie@pku.edu.cn

Shikun Zhang
Peking University
Beijing, China
zhangsk@pku.edu.cn

Wei ye
Peking University
Beijing, China
wye@pku.edu.cn

Abstract

Unit tests play a vital role in the software development lifecycle. Recent advances in Large Language Model (LLM)-based approaches have significantly improved automated test generation, garnering attention from both academia and industry. We revisit LLM-based unit test generation from a novel perspective by decoupling prefix generation and assertion generation. To characterize their respective challenges, we define Initialization Complexity and adopt Cyclomatic Complexity to measure the difficulty of prefix and assertion generation, revealing that the former primarily affects compilation success, while the latter influences test coverage.

To address these challenges, we propose *Seed&Steer*, a two-step approach that combines traditional unit testing techniques with the capabilities of large language models. *Seed&Steer* leverages conventional unit testing tools (e.g., EvoSuite) to generate method invocations with high compilation success rates, which serve as seeds to guide LLMs in constructing effective test contexts. It then introduces branching cues to help LLMs explore diverse execution paths (e.g., normal, boundary, and exception cases) and generate assertions with high coverage. We evaluate *Seed&Steer* on five real-world Java projects against state-of-the-art baselines. Results show that *Seed&Steer* improves the compilation pass rate by approximately 7%, successfully compiling 792 and 887 previously failing cases on two LLMs. It also achieves up to 73% branch and line coverage across focal methods of varying complexity, with coverage improvements ranging from $1.09\times$ to $1.26\times$. Our code, dataset, and experimental scripts will be publicly released to support future research and reproducibility.

Keywords

Unit Test Generation, Large Language model,

ACM Reference Format:

Shuaiyu Zhou, Zhengran Zeng, Xiaoling Zhou, Rui Xie, Shikun Zhang, and Wei ye. 2025. Seed&Steer: Guiding Large Language Models with Compilable Prefix and Branch Signals for Unit Test Generation. In . ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

1 Introduction

Unit testing is a key part of modern software development, used to check individual methods or functions before system integration [24, 26, 36]. However, manually writing and maintaining these tests is time-consuming and error-prone [10, 21], especially as codebases evolve. To reduce this workload, researchers have developed automated test generation techniques—including search-based [8, 11, 19, 6, 12, 15, 18], constraint-based [9, 13, 32, 22, 28], and random-based [35, 25, 5] approaches—which mainly aim to maximize code coverage.

However, these traditional methods often create tests that are hard to read, understand, or reuse, limiting their practical adoption [3, 16]. Language model-based approaches have emerged as a promising alternative. Using pretrained Transformer architectures [30], tools like A3test [2] and AthenaTest [29] can learn from human-written tests to generate cases that are structurally correct and logical. Furthermore, large language models like ChatGPT [1] show a strong ability to generalize for code understanding and generation, approaching human-level comprehension [14, 23, 27]. Similarly, LLM-based tools such as TestGen-LLM [4], [7], ChatUniTest [33], and ChatTESTER [34] can produce easy-to-understand tests with good accuracy and coverage. These LLM-based approaches have significantly advanced unit test generation, attracting widespread attention in both academia and industry.

Unit test generation consists of two main phases: test prefix generation and assertion generation. The test prefix sets up the testing environment by building the test class structure, initializing variables, and configuring object states to ensure the target method runs correctly. Assertion generation verifies the method's correctness by creating conditions that compare actual results with expected outcomes. In this work, we examine LLM-based unit test generation from a novel, finer-grained perspective by decomposing the process into these two distinct stages. We define Initialization Complexity and adopt Cyclomatic Complexity to assess the difficulty of prefix and assertion generation. Through preliminary studies (detailed in Section 2), we reveal that test prefix generation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference'17, Washington, DC, USA

© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

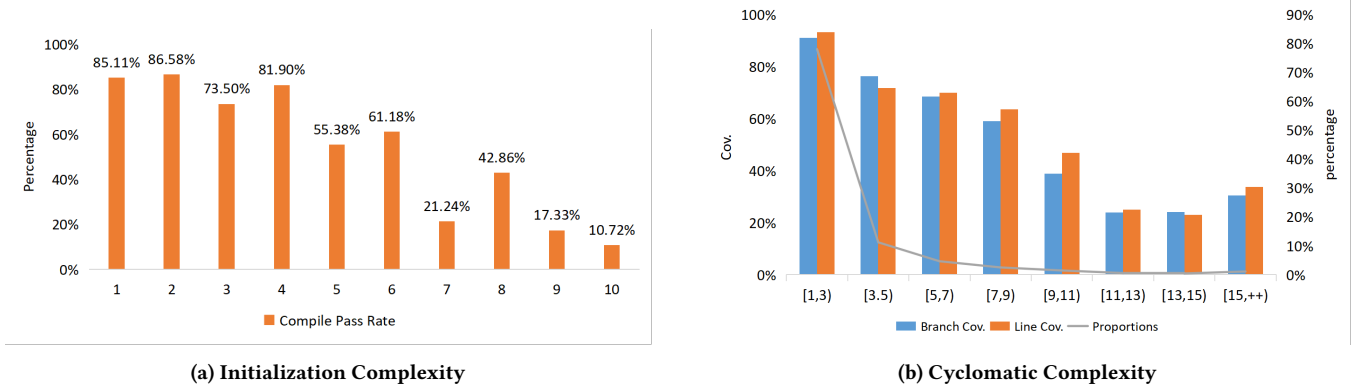


Figure 1: (a)With higher initialization complexity, LLMs struggle to invoke the focal method correctly. Correct invocation examples may help LLMs learn to generate compilable prefixes. (b)As cyclomatic complexity grows, simple instructions fail to convey code context. Guiding LLMs with more branch information may lead to deeper coverage.

and assertion generation have distinct impacts on the compilability and coverage of the final test cases. Specifically, we present two key findings:

(1) **The lack of proper method invocation in the prefix generation stage is a major contributor to compilation failures.** Many real-world methods require complex initialization contexts, such as nested object creation, precondition configuration, or specific method invocation sequences. Due to limited understanding of domain-specific method usage, LLMs often generate invalid test prefixes, leading to uncompileable code. As discussed in Section 2.1, we found that the higher the initialization complexity (see formula 1) of a method, the more difficult it is for LLMs to generate compilable tests, with success rates dropping to as low as 10.72% in the most complex cases.

(2) **Complex code logic causes generated assertions to be superficial, leading to poor branch and line coverage.** Existing prompting strategies often rely on generic instructions such as "write a test case for this method and follow these rules: ...", which offer only coarse-grained guidance. This makes it difficult for LLMs to reason about fine-grained branching conditions or boundary cases, especially in methods with non-trivial control flow. As discussed in Section 2.2, we found that for methods with complex code logic, LLMs struggle to fully understand the context, resulting in limited coverage of deeper parts of the method, sometimes covering only around 20% code in the worst cases.

In this paper, we present *Seed&Steer*, a novel two-step approach that combines traditional unit testing practices with the capabilities of large language models. In the test prefix generation step, which we named the Seed process, we introduce *EvoSuite*, which analyzes almost every method in a test class, including local methods, based on bytecode. We extract the method invocation cases from them to help LLMs generate unit tests, which compensates for LLMs' limited understanding of class constructors, object lifecycles, and dependency constraints by generating test cases that successfully invoke the focal method. In the assertion generation step (Steering process), we first extract conditional branching statements in the focal method to assist the LLM in generating branching intents, which help the LLM focus on different execution paths (e.g.,

normal, boundary, and exception cases), and finally generate assertions in conjunction with function intents. We conduct extensive experiments on five real-world Java projects and demonstrate that *Seed&Steer* consistently outperforms existing baselines in terms of compilation success, test execution, as well as branch and line coverage.

In summary, our work makes the following key contributions:

- We revisit LLM-based unit test generation from a novel perspective by separating prefix generation and assertion generation, revealing that the former primarily affects compilation success, while the latter influences test coverage.
- We propose *Seed&Steer*, a simple yet effective method that combines traditional tools with LLMs to construct compilable test prefixes and generate branch-guided assertions, achieving promising results in compilation success, test execution, and code coverage.
- Our findings categorize the limitations of LLM-based automated unit test generation into two concrete complexity challenges, which offers a unique perspective for future research.

We will release all code, datasets, and experiment scripts ¹ to support reproducibility and foster future research in LLM-based unit test generation.

2 Preliminary Study

We found that there are always limitations when generating unit tests for complex focal methods. To better illustrate the challenges we found, we quantified the impact of **Initialization Complexity** and **Cyclomatic Complexity** on the existing method *ChatTester* [34].

2.1 Initialization Complexity

To quantify the impact of initialization logic on test generation effectiveness, we define four static features: the number of variable declarations V , the number of object creations O , the number of method calls before invoking the focal method M , and parameter

¹<https://github.com/monster29000/SeedandSteer>

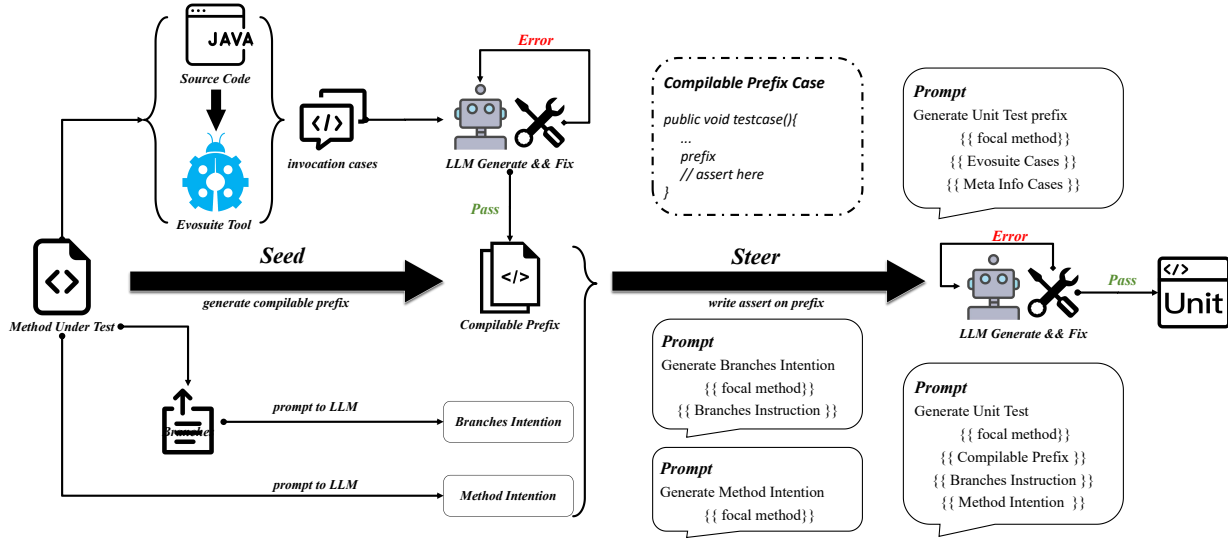


Figure 2: Overview of the Seed&Steer workflow.

number P . Based on these, We define the overall **Initialization Complexity Score** as a weighted sum of four normalized static features:

$$\text{Initialization Complexity} = w_1 \cdot \hat{V} + w_2 \cdot \hat{O} + w_3 \cdot \hat{M} + w_4 \cdot \hat{P} \quad (1)$$

Here, \hat{x} denotes the min-max normalized value of feature x , and w_1, w_2, w_3, w_4 represent the corresponding weights. Through correlation analysis, we observed that the number of method calls before invoking the focal method M and the number of parameters P are more strongly negatively correlated with compilation success. Therefore, we assign higher weights to these two factors in the final complexity score, resulting in a weight configuration of $[0.1, 0.1, 0.4, 0.4]$.

As shown in Figure 1a, we scale the Initialization Complexity Score into bins (0–1, 1–2, ..., 9–10) and report the compilation pass rate within each bin. The trend is clear: as the initialization complexity increases, the proportion of tests that are successfully compiles drops significantly. For instance, tests with complexity scores between 1 and 2 achieve a pass rate of 86.58%, while those in the 9–10 range drop to only 10.72%. This observation highlights that initialization complexity is a major obstacle in generating compilable test code. This suggests that the complex initialization critical obstacle to compilation success.

We find that complex initialization is a key limiting factor in automated test generation. As initialization becomes more intricate, LLMs often fail to correctly invoke the focal method, leading to compilation failures.

2.2 Cyclomatic Complexity

Beyond initialization logic, we also investigate the impact of focal method structural complexity on test generation. We compute the

Cyclomatic Complexity Number (CCN) of each focal method using the static analysis tool Lizard, and correlate it with coverage metrics.

As shown in Figure 1b, we observe that approximately 85% of methods have a CCN between 1 and 3, typically representing simple constructors or utility functions. These low-complexity methods are generally well-handled by state-of-the-art large language models. In contrast, methods with CCN values ranging from 4 to 10 often containing intricate control flow and business logic-exhibit a sharp decline in test coverage, with both branch and line coverage typically falling below 50%.

We find that method complexity is a limiting factor in automated test generation. As code logic becomes more complex, large language models often fail to fully understand the code context, resulting in lower code coverage.

3 Methodology

This section presents a detailed introduction to the proposed method, **Seed&Steer**. The goal of **Seed&Steer** is to enhance the effectiveness of LLMs in unit test generation, whose core idea is to divide the test generation process into two key stages. In the first stage, we collect representative invocation cases of the focal method and construct high-quality, compilable test prefixes that serve as a stable and semantically meaningful foundation for subsequent test generation. In the second stage, we extract conditional branch signals from the focal method to derive branch intents, which are then integrated with functional intents to formulate more precise and context-aware prompts. These prompts guide the model in generating assertions with improved semantic alignment and higher code coverage.

In the following subsections, we initially elaborate on the extraction of meta information from static source code in Section 3.1

and then discuss the construction of compilable test prefixes in Section 3.2. Furthermore, Section 3.3 introduces our strategy for leveraging branching intents to guide the generation of high-quality assertions. Finally, Section 3.4 details the repair mechanism and the iterative generation strategy designed to enhance test effectiveness.

3.1 Code Preprocessing

Before initiating the test generation process, we conduct a static parse over the source code to extract structured meta information that will be utilized in subsequent stages, such as prefix construction and branch guidance.

Specifically, we traverse each Java file in the target project and parse it into an Abstract Syntax Tree using the `javalang` and `tree-sitter` libraries in Python. This parsing enables us to systematically extract class-level information, including class name, type (e.g., interface, abstract, or concrete), inheritance relations, source code content, field declarations, and all method definitions.

We formalize a class as a structured object C , defined as:

$$C = \{\text{name, type, superclass, content, fields, } \mathcal{M}\},$$

where \mathcal{M} denotes the set of methods declared in the class. We further partition \mathcal{M} into two subsets:

- $\mathcal{M}_{\text{focal}}$: Public methods that are considered as targets.
- $\mathcal{M}_{\text{other}}$: Non-focal methods such as private or utility.

A focal method is defined as a public method in a public, non-abstract class. Accordingly, we define the focal method set \mathcal{F} as follows:

$$\mathcal{F} = \{f_i = (\text{name, params, invocation, signature, content}) \mid f_i \in \mathcal{M}_{\text{focal}}\}.$$

Each focal method f_i contains its name, parameter list, invocation information (i.e., which methods it calls), full signature, and the corresponding method code. The "invocation" in the formula is used to filter the correct call cases of the function, the "signature" is utilized to filter the focal method when calculating the coverage, and the content is employed to extract the conditional branching statements. This structured metadata is essential for building meaningful prompts and constructing compilable test prefixes in the downstream test generation pipeline.

This systematic preprocessing process ensures that the model is provided with precise and complete contextual information, which serves as the foundation for the subsequent **steps**: *Seed* and *Steer*.

3.2 Seed: Compilable Prefix

To generate high-quality test prefixes for a given focal method $\mathcal{M}_{\text{focal}}$, we design a seed generation strategy that combines static parse with example extraction. This strategy leverages both automatically generated tests from EvoSuite and manually discovered calling patterns from the source class C to ensure realistic and executable prefix construction.

Generating EvoSuite test set \mathcal{T}_{ES} . We begin by applying EvoSuite, a bytecode-based unit test generator, to class C that contains the focal method. EvoSuite generates a compilable and structurally valid test class, where each generated test class is processed into a structured representation:

$$\mathcal{T}_{ES} = \{\text{name, type, content, fields, } \mathcal{M}_{tc}\},$$

where \mathcal{M}_{tc} denotes the set of test methods, each of which may include one or more method invocation traces. All test code is preprocessed using the same procedure described in Section 3.1.

Subsequently, to enhance the executability and realism of the prefixes, we adopt a dual-path strategy:

Path 1: Source-level exemplar mining. We traverse all methods in the source class, denoted by $\mathcal{M} = \mathcal{M}_{\text{focal}} \cup \mathcal{M}_{\text{other}}$. For each method $m \in \mathcal{M}$, if the method's invocation list contains a call to $\mathcal{M}_{\text{focal}}$, we extract the object instantiation and argument preparation statements as a candidate prefix template.

To further improve diversity, we additionally examine the EvoSuite-generated test methods \mathcal{M}_{tc} , and if any method invokes $\mathcal{M}_{\text{focal}}$, we extract at most three such test cases.

Path 2: EvoSuite-only mining when source lacks calls. If no method in \mathcal{M} directly invokes the focal method, we fallback to the EvoSuite test cases. We iterate over all $m \in \mathcal{M}_{tc}$ in \mathcal{T}_{ES} and extract up to five examples where the focal method is invoked.

Constructing LLM prompts and generation. We then construct a prompt that contains the extracted positive examples and explicitly instruct the LLM to generate only the test prefix that correctly invokes $\mathcal{M}_{\text{focal}}$, without assertions. Upon generation, we remove all assertion statements and inject comments (e.g., `// TODO: assert here`) to mark where an assertion would be added.

Iterative refinement via compilation feedback. The generated code is compiled using a local Java compiler. If the compilation fails, we capture the error messages and construct new prompts containing the diagnostics to guide the LLM in fixing the code. This process is repeated for at most five iterations to control resource usage and token budget.

3.3 Steer: Branch Intentions

After obtaining a compilable test prefix $\mathcal{P}_{\text{pass}}$, the next step **Steer** is to guide the LLM to generate meaningful and effective assertions. This phase focuses on maximizing code coverage by steering the model with structured semantic and control-flow information.

We first perform a static parse on the focal method $\mathcal{M}_{\text{focal}}$ to identify key branching structures. These include:

- Conditional branches: `if`, `else if`, `switch`.
- Loop constructs: `for`, `while`, `do-while`.
- Exception handling: `try-catch`, `throw`.
- Input-dependent statements.

The extracted set of such branch points is denoted as $\mathcal{B}_{\text{cond}}$. For each $b_i \in \mathcal{B}_{\text{cond}}$, we prompt the LLM to infer and articulate its triggering conditions in natural language. For instance, a branch like `if (a > 0 && b == null)` would yield an intention such as: *"This branch is taken when the first argument is positive and the second is null."* These elements collectively contribute to the formation of the branch intention $\mathcal{I}_{\text{branch}}$.

Beyond branch-level semantics, we further summarize the overall intent of the focal method through a function-level intention $\mathcal{I}_{\text{func}}$, benefiting from the success of *ChatTester* [34]. This includes the method's purpose, input-output behavior, side effects, and corner cases.

The final prompt provided to the LLM is constructed by concatenating the following components:

- The verified test prefix $\mathcal{P}_{\text{pass}}$.

- Branch intentions $\mathcal{I}_{\text{branch}} = \{b_i\}_{i=1}^n$.
- The function intention summary $\mathcal{I}_{\text{func}}$.

This structured guidance enables the LLM to generate the final test case \mathcal{T}_{gen} with precise and semantically aligned assertions. In contrast to vague natural language instructions, our approach grounds assertion generation in concrete program semantics, thereby enhancing both coverage and correctness.

3.4 Iterative Error Correction

After generating a complete test case, we first apply lightweight syntactic and structural corrections, including inserting missing import statements, renaming classes or methods, and resolving common formatting issues. This preprocessing step ensures that the test case is syntactically valid and compilable.

Subsequently, we compile and execute the generated test case in accordance with the procedure outlined in [34]. In the event of compilation or runtime errors, we utilize the resulting error feedback to reconstruct the prompt and re-invoke the LLM for correction. This iterative refinement process continues until the errors are resolved or a predefined maximum number of attempts δ is reached.

The errors are categorized into two types:

- **Compilation Errors** ($\mathcal{E}_{\text{compile}}$): Usually caused by incorrect variable usage, method signatures, or misinterpreted seed code. We extract the compiler’s diagnostic messages to construct targeted prompts for regeneration.
- **Runtime Errors** ($\mathcal{E}_{\text{runtime}}$): These include assertion failures or exceptions such as `NullPointerException`. When assertions fail, we treat them as candidate signals for revision. Moreover, if the error persists after δ attempts, we mark the test as partially valid and still record its coverage.

Furthermore, we formalize the correction process as a repair function:

$$\text{Fix}(\mathcal{T}, \mathcal{E}, \delta) \rightarrow \mathcal{T}^* \text{ or } \perp$$

where \mathcal{T} is the original test, \mathcal{E} denotes error feedback, and \mathcal{T}^* is the corrected test. If all δ attempts fail, we return \perp to indicate unrecoverable failure.

In cases where the three-stage pipeline,

Seed Construction \rightarrow *Intention Steer* \rightarrow *Iterative Generate&Repair*

fails to generate a valid test, we fall back to a two-stage strategy:

Seed Template & Intention Steer \rightarrow *Iterative Generate&Repair*

This feedback-driven refinement process significantly improves test success rates and reduces reliance on manual inspection.

4 Experiments

4.1 Datasets

In this section, we first introduce the construction and statistics of the evaluation dataset. Afterwards, we introduce the configuration for generating tests and baselines.

Dataset. Defects4J[20] is a widely used benchmark suite that provides real-world software bugs and supporting infrastructure for evaluating software engineering techniques. Following prior work such as ChatUniTest[33], TestART[17] and HITS[31], we select five representative Java projects from Defects4J[20] to evaluate

Table 1: Projects dataset

Project	Version	Focal classes	Focal methods
Gson	2.10.1	52	378
Commons-Lang	3.1.0	87	1728
Commons-Cli	1.6.0	20	177
Commons-Csv	1.10.0	8	137
JFreeChart	1.5.4	499	5772
Total		666	8192

our approach *Seed&Steer*, as summarized in Table 1. We utilize the publicly released pickle-format dataset and project environments provided by TestART, which include source code and corresponding pom.xml Maven configuration files. From these environments, we extract all public, non-abstract classes and identify their public methods as focal methods, resulting in a total of **8,192** focal methods across **666** classes.

The data we used is not like *A3test*[2] or *ChatTester* [34] which are in the framework of Defects4J to realize mass runs with different buggy versions of the same project. We chose the same 5 Java projects as them, but fixed the specific versions so that we could compile, execute and evaluate them efficiently. And with a fixed project environment, the EvoSuite-based unit test generation process is easy and repeatable.

Complex Methods. To comprehensively evaluate the performance of *Seed&Steer* on more challenging test generation tasks, we categorize methods under test based on their static complexity and focus on cases with higher structural complexity, as show in Figure 1b.

4.2 Baselines

In order to evaluate the effectiveness of our proposed *Seed&Steer* method, we use the state-of-the-art unit test generation methods using an LLM as our baseline. Specifically, we choose base model (ChatGPT-3.5, ChatGPT-4.0), ChatUniTest, ChatTester and TestART as our baseline.

ChatUniTest. ChatUniTest is a recent LLM-based test generation framework built upon ChatGPT. It adopts a generation-validation-repair paradigm. ChatUniTest is one of the most prominent prompt-based LLM testing frameworks and demonstrates the potential of combining static analysis with in-context learning.

ChatTester. ChatTester splits the test generation task into two subtasks firstly understanding the intent of the core method and generating the corresponding unit tests designed to generate tests with higher quality assertions. This is followed by iteratively fixing compilation errors generated during the initial test generation process, maximizing the use of compiler error information to guide LLM.

TestART. TestART is an approach to improve LLM-based unit test through the co-evolution of automated generation and repair iteration. TestART leverages the generative capabilities of LLMs by integrating generation-repair co-evolution, testing feedback and prompt injection into the iteration, repairing the bugs contained in the generated test cases and feeding back the coverage information for outputting high-quality test cases.

Table 2: Effectiveness comparison of Seed&Steer and baseline approaches across four standard metrics.

Method	Projects	Focal methods	Compile passed Rate	Test Passed Rate	Branch Coverage	Line Coverage
ChatGPT-3.5			74.39%	49.91%	43.10%	42.58%
ChatGPT-4.0			80.50%	59.75%	51.86%	50.88%
ChatUnitTest			76.52%	60.05%	48.68%	47.39%
TestART			87.05%	78.55%	69.40%	68.17%
ChatTester _{gpt-3.5-turbo}	Total	8192	85.88%	57.18%	50.80%	48.68%
ChatTester _{gpt-4o}			87.21%	52.12%	52.78%	53.52%
Seed&Steer _{gpt-3.5-turbo}			<u>92.77%</u>	<u>69.87%</u>	<u>72.19%</u>	<u>71.20%</u>
Seed&Steer _{gpt-4o}			95.80%	69.34%	73.30%	75.26%
Seed&Steer _{gpt-3.5-turbo}	Gson	378	83.33%	56.61%	69.62%	66.00%
	Lang	1728	88.60%	67.36%	65.93%	66.46%
	Cli	177	85.88%	62.71%	63.57%	65.69%
	Csv	137	70.00%	49.64%	65.38%	72.19%
	Chart	5772	95.39%	72.19%	75.55%	72.91%
Seed&Steer _{gpt-4o}	Gson	378	94.71%	68.78%	76.77%	82.06%
	Lang	1728	97.74%	77.84%	77.60%	81.19%
	Cli	177	94.92%	66.10%	88.10%	89.06%
	Csv	137	96.35%	79.56%	76.75%	78.35
	Chart	5772	94.65%	66.68%	71.15%	72.70%

4.3 Experiment Setup

In our evaluation, *Seed&Steer* performs up to five iterations for each focal method. Within each iteration, it allows at most five rounds of seed repair and five rounds of test case repair. If a generated test case contains compilation or runtime errors that cannot be resolved within the current iteration, the process is terminated early to avoid unnecessary computation. To ensure fair comparison, *Seed&Steer* and all baseline methods—including *TestART*—use the *gpt-3.5-turbo* API, which supports up to 16K tokens of context. For experiments involving *gpt-3.5-turbo* and *gpt-4o*, we adopt a unified three-stage procedure: test generation, compilation and execution, and iterative error correction. All hyperparameter settings and prompt formats are kept consistent with those used in *Seed&Steer*. The evaluation results of *ChatUnitTest* are obtained from the official results reported in *TestART*. During test execution, we use Java 8 as the compilation and runtime environment, JUnit 4 as the unit testing framework, and Jacoco² to collect branch and line coverage metrics.

5 Evaluation And Results

5.1 Research Questions

We raise the following research questions to evaluate the effectiveness of our method.

- **RQ1 Effectiveness Comparison.** How is the performance of Seed&Steer compared to state-of-the-art (SOTA) approaches?

- **RQ2 Ablation Comparison.** Are the components of seed and steer necessary for the success of Seed&Steer?
- **RQ3 Compilation Enhancement by Seed.** Does the seed construction strategy in Seed&Steer significantly improve the compilation success rate, particularly in addressing cases where an insufficient invocation context causes failures?
- **RQ4 Coverage Boost by Steer.** How much does the branch-guided steer strategy contribute to test coverage, and can it consistently improve both branch and line coverage across different levels of method complexity?

5.2 Answer to RQ1: Effectiveness Comparison

We evaluated the overall effectiveness of *Seed&Steer* by comparing it with a set of representative baselines, including *ChatUnitTest*, *ChatTester*, *TestART* and raw *ChatGPT-3.5* & *ChatGPT-4.0*. We report results across multiple dimensions: compilation pass rate, test execution pass rate, branch coverage, and line coverage.

We compare *Seed&Steer* with seven baseline approaches in key evaluation metrics. As shown in Table 2, we first report the average performance in the full data set, followed by a detailed breakdown of the results in five individual Java projects.

Our method demonstrates a significant advantage in terms of compilation pass rate. **Seed&Steer’s compilation pass rates for calling *gpt-3.5-turbo* and *gpt-4o* models are 92.77% and 95.80%, respectively.** The direct call to *gpt-3.5-turbo* (*ChatGPT-3.5*) compiles with a pass rate of only 74.39%, while *Seed&Steer(gpt-3.5-turbo)* improves the compilation pass rate to 92.77%. Also calling the *gpt-3.5-turbo* model, *Seed&Steer(gpt-3.5-turbo)* outperforms

²<https://github.com/jacoco/jacoco>

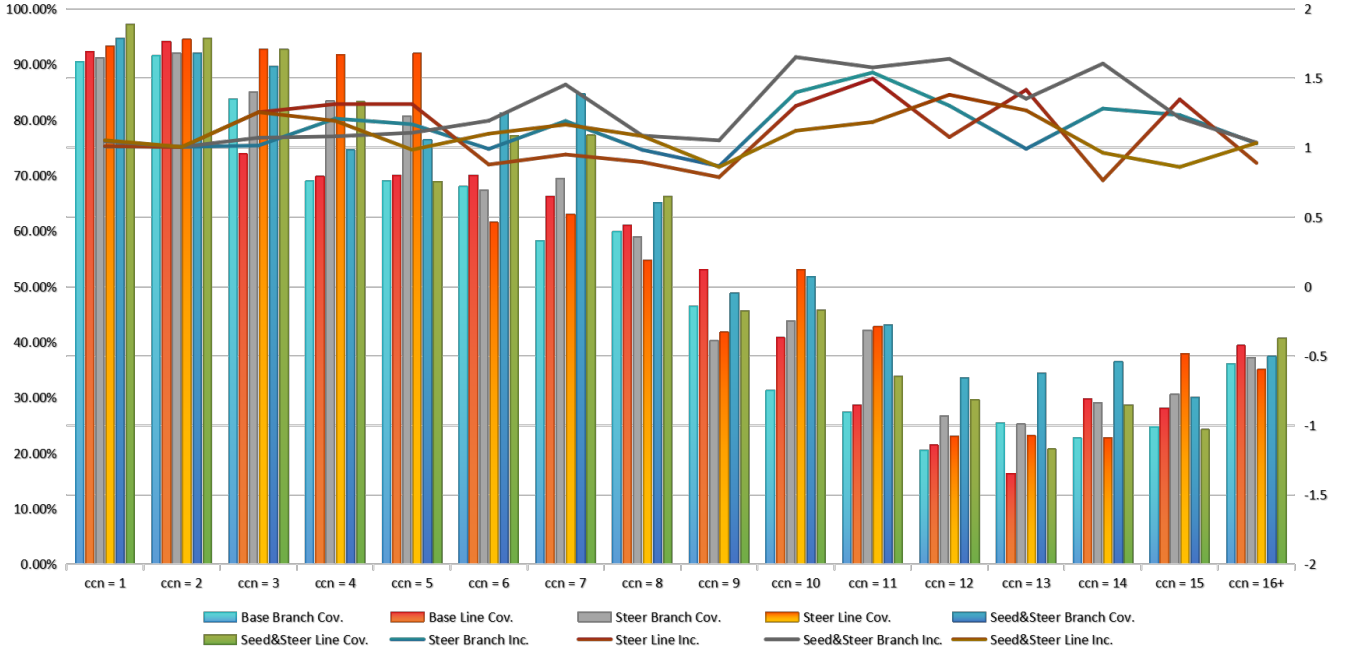


Figure 3: Coverage Improvements from Steer Across Methods of Varying Cyclomatic Complexity

Table 3: Ablation study of *Seed&Steer* under two LLM backends (gpt-3.5-turbo and gpt-4o).

Variant	Seed&Steer _{gpt-3.5-turbo}				Seed&Steer _{gpt-4o}			
	Compile Pass	Test Pass	Branch Cov.	Line Cov.	Compile Pass	Test Pass	Branch Cov.	Line Cov.
VanillaLLM (ChatTester)	85.88%	57.18%	50.80%	48.68%	87.21%	52.12%	52.78%	53.52%
+Seed Only	95.54%	70.47%	59.70%	61.09%	98.03%	59.55%	48.82%	47.37%
+Steer Only	87.95%	63.89%	<u>61.65%</u>	<u>65.91%</u>	85.68%	50.74%	<u>68.09%</u>	<u>68.71%</u>
Seed&Steer (full)	<u>92.77%</u>	<u>69.87%</u>	72.19%	71.20%	<u>95.80%</u>	69.34%	73.30%	75.26%

TestART (+5.72%), ChatTester (+6.89%), and ChatUniTest (+16.25%). Similarly, under gpt-4o model, Seed&Steer(gpt-4o) achieves 15.30% and 8.59% improvements over ChatGPT-4.0 and ChatTester, respectively.

In terms of test execution pass rate, *Seed&Steer* also achieves competitive results, reaching 69.87% with gpt-3.5-turbo and 69.34% with gpt-4o. These values significantly outperform ChatGPT-3.5 (by 20%), ChatGPT-4.0 (by 10%), and ChatTester (by 15%). Although there is still a 9% gap compared to TestART (78.55%), *Seed&Steer* remains competitive in overall reliability.

For branch and line coverage, *Seed&Steer* delivers strong performance. When using gpt-3.5-turbo, it achieves 72.19% branch and 71.20% line coverage. With gpt-4o, the scores are 73.30% and 75.26%, respectively. Compared to the second-best method, TestART (69.40% and 68.17%), Seed&Steer(gpt-3.5-turbo) yields absolute gains of +2.79% and +3.03%. The improvements over ChatGPT-3.5, and ChatUniTest are even more pronounced—approaching 30%, and 25%, respectively. Notably, Seed&Steer(gpt-4o) achieves the

best overall performance by leveraging the stronger generation capabilities of gpt-4o.

Finding 1

Seed&Steer outperforms existing baselines in terms of compilation pass rate, branch coverage, and line coverage. By leveraging compilable test prefixes and branch-intention-guided assertion generation, it demonstrates superior performance across multiple evaluation metrics.

5.3 Answer to RQ2: Ablation Comparison

To better understand the contribution of each component in *Seed&Steer*, we conduct an ablation study by selectively disabling the seed construction and steer guidance mechanisms. Specifically, we evaluate the following two variants: (1) *SeedOnly*, which uses EvoSuite-derived seeds but without branch-guided steer prompts; (2) *SteerOnly*,

Table 4: Compilation Gains With vs. Without Invocation

Model	Method	With Invocation			Without Invocation		
		Invocation num	Pass num	Compile Pass	No Invocation num	Pass num	Compile Pass
gpt-3.5-turbo	ChatTester _{gpt-3.5-turbo}	3610	3038	84.16%	4582	3997	87.23%
	SeedOnly		+396	95.12%		+396	95.88%
	Seed&Steer _{gpt-3.5-turbo}		+284	92.02%		+281	93.37%
gpt-4o	ChatTester _{gpt-4o}	3610	3073	85.12%	4582	4071	88.85%
	SeedOnly		+480	98.42%		+407	97.73%
	Seed&Steer _{gpt-4o}		+350	94.82%		+354	96.57%

which retains the branch-guided prompts but omits seed initialization. In fact the strategy that neither component is used is *ChatTester*. We compare these variants with the full (3) *Seed&Steer* pipeline across all key metrics to answer RQ2. The results are displayed in Table 3

(1) **SeedOnly**. Adding the Seed component improves compilation pass rates for both models: from 85.88% to 95.54% on gpt-3.5-turbo, and from 87.21% to 98.03% on gpt-4o. The test execution pass rate also increases notably, from 57.18% to 70.47% and from 52.12% to 59.55%, respectively. However, coverage rates on gpt-4o decrease slightly with Seed: branch coverage drops from 52.78% to 48.82%, and line coverage from 53.52% to 47.37%.

(2) **SteerOnly**. The Steer component leads to substantial improvements in coverage. On gpt-3.5-turbo, branch and line coverage increase from 50.80% and 48.68% to 61.65% and 65.91%, respectively. On gpt-4o, branch coverage improves from 52.78% to 68.09%, and line coverage from 53.52% to 68.71%. Meanwhile, Steer slightly reduces the compilation pass rate: on gpt-3.5-turbo, it drops from 85.88% to 87.95%, and on gpt-4o, from 87.21% to 85.68%.

(3) **Seed&Steer**. Combining both components yields improvements across most metrics. For gpt-3.5-turbo, the compilation pass rate rises to 92.77%, and branch and line coverage increase to 72.19% and 71.20%, outperforming both Seed-only and Steer-only variants. The test pass rate (69.87%) is slightly below that of Seed alone (70.47%) but remains comparable. In gpt-4o, compilation success reaches 95.80% - higher than Steer (85.68%) but lower than Seed (98.03%). Coverage improves to 73.30% (branch) and 75.26% (line), exceeding both the base and *SeedOnly* methods, and only slightly trailing the *SteerOnly* peak values.

5.4 Answer to RQ3: Compilation Enhancement

To more deeply understand the role of the *seed* component in improving compilation success, we categorize the focal methods into two groups: those containing explicit method invocations and those without. We then evaluate compilation outcomes across three variants—base model, *SeedOnly*, and the full *Seed&Steer* pipeline on both gpt-3.5-turbo and gpt-4o.

As show in Table 4, for two base model method, the compilation pass rate is 84.16% & 85.12% for methods with invocations and 87.23% & 88.85% for those without. Interestingly, methods lacking invocations exhibit slightly higher success rates, likely because the model tends to generate minimalistic, syntactically valid test cases in such contexts.

With the introduction of the Seed component, both categories witness significant improvements. On gpt-3.5-turbo, compilation rates increase to 95.12% (with-invocation) and 95.88% (no-invocation), successfully recovering 792 previously failed cases, which accounts for 9.67% of the dataset. A similar or stronger trend is observed with gpt-4o, where compilation rises from 85.12% to 98.42% (with-invocation) and from 88.85% to 97.73% (no-invocation), fixing 887 previously failed cases. This validates that Seed component provides richer cases for local methods that contain invocation, and additional cases for local methods that don't contain invocation, and the compilation pass rate is greatly improved.

Upon full pipeline, we observe a slight decline in compilation success. For gpt-3.5-turbo, the rate drops from 95.54% (*SeedOnly*) to 92.77% (full pipeline), and for gpt-4o, from 98.03% to 95.80%. This reduction is expected, as the steer mechanism introduces some prompts that may increase the likelihood of compilation errors. Nevertheless, *Seed&Steer*'s compilation pass rate is still substantially better than the original model, suggesting that the structural prefixes provided by *Seed* have continued to support overall stability.

Finding 2

Ablation studies on the two components of *Seed&Steer* reveal that each individually contributes to improving LLM performance—compilable seeds enhance compilation pass, while branch-guided assertions improve coverage. However, combining both components can lead to a decrease in compilation pass rate.

Finding 3

Regardless of whether the original class contains an invocation of the focal method, the *Seed* component enriches method invocation examples and helps the LLM compile more test cases. Specifically, it resolves 792 and 887 previously failed cases on the two models, respectively.

5.5 Answer to RQ4: Coverage Boost

In our ablation study, we observe that the Steer component significantly improves both branch and line coverage. To further understand *Steer*'s contribution, we grouped focal methods by their cyclomatic complexity number (CCN) using static analysis from Lizard. As shown in Figure 3, *Steer* consistently improves both coverage metrics across all CCN levels under gpt-3.5-turbo. The relative improvement ranges from $1.09\times$ to $1.26\times$, with more significant gains observed for high-complexity methods (e.g., $\text{CCN} > 9$). While methods with CCN between 1 to 4 typically see strong performance from all approaches, more complex cases (e.g., CCN in 11-14) show notable degradation in baseline performance, whereas *Steer* still achieves up to $1.6\times$ coverage improvement.

Compared to the full *Seed&Steer* pipeline, using *Steer* alone yields more stable and consistent gains across complexity levels. Further analysis of newly covered branches reveals that most of them correspond to deep conditional structures and boundary-specific behaviors, indicating that the intent-driven guidance provided by *Steer* effectively helps LLMs generate higher-quality assertion statements.

Finding 4

The *Steer* component provides branch-intent context, which enables deeper code coverage even for methods with high cyclomatic complexity, achieving an increase in coverage ranging from $1.09\times$ to $1.26\times$.

5.6 Threats to Validity

LLM Internal Selection. Our experiments are conducted under consistent settings, using the same API calls (gpt-3.5-turbo and gpt-4o) and context lengths across all baselines. However, the inherent randomness of LLMs (e.g., temperature or sampling strategies) may influence the generation results. We mitigate this by repeating each test multiple times and reporting averaged outcomes.

Datasets Selection. Our evaluation is limited to five real-world Java projects. Although these projects span a diverse range of complexities and code patterns, they may not fully represent other programming languages or software domains. Future work could explore the generalizability of *Seed&Steer* to languages such as Python or C++, or to industrial-scale codebases.

Metrics Selection. We adopt four widely used metrics: compilation pass rate, test execution pass rate, branch coverage, and line coverage to measure effectiveness. While these metrics are standard in test generation research, they do not directly assess semantic correctness or the adequacy of the generated tests. Future extensions may incorporate human-in-the-loop evaluation or semantic-aware metrics.

6 Discussion

6.1 Execution Overhead.

Seed&Steer relies on EvoSuite to generate seed tests during initialization. Although this introduces moderate latency when generating the first test for a target class, it remains within a reasonable range compared to other baselines. As shown in Table 5, *Seed&Steer*

exhibits slightly longer average per-method time costs compared to ChatTester (5.8s increase under GPT-3.5 and 3.6s under GPT-4o). However, this overhead is acceptable given the enhanced capability of *Seed&Steer* in handling complex initialization contexts and hard-to-cover code paths that are often missed by baseline pipelines. In practice, methods without EvoSuite support often suffer from longer total correction time due to repeated generation of uncompileable code. By providing syntactically and semantically valid seeds upfront, *Seed&Steer* effectively mitigates this problem.

We do not compare our approach to EvoSuite's own metrics because its goal is to maximize coverage for the entire class, whereas our approach is oriented towards a specific focal method and *Steer*'s operations cannot be applied to a class.

6.2 Model Generalization

Beyond evaluating our approach on proprietary large language models such as OpenAI's GPT, we also investigate whether the *Seed&Steer* pipeline generalizes to open-source LLMs. Specifically, we apply our method to Qwen2.5-Coder, a representative open-source code model family, using both the 7B and 32B versions. As shown in Table 6, our approach consistently outperforms the *ChatTester* [34] baseline across multiple metrics, including compile success rate and coverage. These results demonstrate that *Seed&Steer* is not tightly coupled to closed-source models or specific architectures. Instead, it offers a transferable testing generation strategy that is effective even when applied to widely available open-source models.

6.3 Tool dependency

A known limitation of *Seed&Steer* lies in EvoSuite's strict dependency on Java 8. While Java 8 remains prevalent in many legacy and enterprise systems due to its long-term support, it may not align with modern development environments. However, this constraint does not stem from the design of *Seed&Steer* itself. Thanks to its modular architecture, *Seed&Steer* can be easily extended to support newer Java versions by substituting EvoSuite with compatible test generation tools. This flexibility ensures broader applicability across diverse software ecosystems.

And our work demonstrates that it is possible for the output content of traditional tools to be adopted by LLM, and that even without using EvoSuite, the researcher can try to solve problems that we cannot solve on other traditional tools.

7 Related Work

7.1 Automated Unit Test Generation

Automated unit test generation has been a longstanding research topic aimed at alleviating the cost and effort of manual test writing [10, 21]. Traditional approaches fall into three main categories: search-based, constraint-based, and random-based techniques. Search-based methods apply evolutionary algorithms or heuristics to maximize code coverage, as seen in works like EvoSuite [15], TestFul [6], and others [8, 11, 19, 12, 18]. Constraint-based approaches such as DySy [9], GRT [22], and Daikon [13] generate tests by inferring program invariants or solving input constraints. Random-based techniques, like Randoop [25] and fuzz testing [35], generate inputs randomly and rely on statistical exploration. Despite their

Table 5: Per-method time cost (in seconds) under different configurations.

Project	Seed&Steer _{gpt-3.5-turbo}	ChatTester _{gpt-3.5-turbo}	Seed&Steer _{gpt-4o}	ChatTester _{gpt-4o}
Gson	27.45 [12.18, 37.49]	20.06 [13.17, 29.33]	25.92 [18.17, 36.16]	19.48 [15.17, 29.72]
Lang	14.38 [9.17, 22.63]	12.62 [11.18, 24.47]	12.92 [11.18, 15.45]	9.25 [8.92, 18.45]
Cli	25.21 [10.87, 32.16]	17.31 [9.87, 29.20]	23.36 [15.87, 32.53]	17.05 [14.87, 29.71]
Csv	38.48 [29.25, 47.39]	28.25 [23.22, 35.62]	15.90 [13.25, 26.82]	10.06 [9.34, 18.07]
JFreeChart	28.04 [23.48, 32.47]	22.56 [13.44, 33.31]	26.31 [22.12, 38.49]	21.28 [15.44, 32.19]
Average	25.25 [9.17, 47.39]	19.43 [9.87, 35.62]	23.24 [11.18, 38.49]	19.68 [9.34, 32.19]

Table 6: Performance using classical open source models

Method	Compile Passed	Test Passed	Branch Cov.	Line Cov.
Seed&Steer _{Qwen2.5-Coder-32B-Instruct}	88.77%	57.51%	55.92%	50.68%
ChatTester _{Qwen2.5-Coder-32B-Instruct}	77.16%	42.36%	44.72%	45.78%
Seed&Steer _{Qwen2.5-Coder-7B-Instruct}	84.42%	48.62%	42.91%	40.87%
ChatTester _{Qwen2.5-Coder-7B-Instruct}	75.13%	34.53%	30.57%	27.64%

effectiveness in achieving high coverage, these methods often produce test cases that are hard to interpret, difficult to maintain, and rarely adopted in practice due to their poor readability and limited semantic alignment with developer intent [3, 16].

7.2 Language Model-Based Unit Test Generation

Recent advances in large language models, especially those based on transformer architectures [30], have sparked a surge in research exploring their capabilities for code generation and understanding [14, 23, 27]. There exist transformer-based approaches, such as AthenaTest[29] and A3Test[2]. ChatUniTest[33] adopts a generation-validation-repair paradigm. ChatTESTER[34] introduce the intention of method, followed by iterative repair guided by compiler errors. TestART[17] further introduces a co-evolutionary loop of test generation and repair, designing some fix templates and integrating feedback signals (e.g., coverage and compilation results) into prompt. HITS[31] semantically splits a method into program slices, guides the LLM to generate unit tests for each slice individually, and finally merges all the tests to improve overall coverage. These incontext-learning pre-processing methods and generic post-processing methods help LLM automated unit test generation. However, these approaches tend to struggle with complex code scenarios, where intricate initialization logic or deeply nested control flows pose challenges to LLMs' understanding and synthesis capabilities.

8 Conclusion

We identified two key challenges faced by LLMs in generating unit tests for complex methods through two preliminary experiments. To solve these two challenged, we propose *Seed&Steer*, a two-phase approach that combines traditional unit testing with the strengths of large language models. In the Seed phase, we leverage EvoSuite to extract method invocation patterns from existing test classes, guiding the LLM to generate valid test prefixes. In the Steer phase,

we extract branching conditions to steer the LLM toward generating diverse test intents and assertions covering different execution paths. We conduct experiments on five real-world Java projects, showing that *Seed&Steer* consistently outperforms existing baselines. In addition, we conducted supplementary experiments to validate the effectiveness and scope of our approach.

We evaluate *Seed&Steer* on five real-world Java projects against state-of-the-art baselines. Experimental results demonstrate that *Seed&Steer* improves compilation pass rates by approximately 7%, successfully compiling 792 and 887 previously failing test cases on two LLMs. It further achieves up to 73% branch and line coverage across focal methods of varying complexity, with relative coverage improvements ranging from 1.09× to 1.26×.

We hope that our approach and preliminary study can offer insights and inspiration for future work in LLM-based test generation.

References

- [1] Josh Achiam et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.
- [2] Saranya Alagarsamy, Chakkrit Tantithamthavorn, and Aldeida Aleti. 2024. A3test: assertion-augmented automated test case generation. *Information and Software Technology*, 176, 107565.
- [3] M Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. 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, 263–272.
- [4] Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. 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*, 185–196.
- [5] James H Andrews, Tim Menzies, and Felix CH Li. 2011. Genetic algorithms for randomized unit testing. *Ieee transactions on software engineering*, 37, 1, 80–94.
- [6] Luciano Baresi and Matteo Miraz. 2010. Testful: automatic unit-test generation for java classes. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering—Volume 2*, 281–284.
- [7] Shreya Bhatia, Tarushi Gandhi, Dhruv Kumar, and Pankaj Jalote. 2024. Unit test generation using generative ai: a comparative performance analysis of autogeneration tools. In *Proceedings of the 1st International Workshop on Large Language Models for Code*, 54–61.

- [8] Arianna Blasi, Alessandra Gorla, Michael D Ernst, and Mauro Pezzè. 2022. Call me maybe: using nlp to automatically generate unit test cases respecting temporal constraints. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 1–11.
- [9] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. 2008. Dysy: dynamic symbolic execution for invariant inference. In *Proceedings of the 30th international conference on Software engineering*, 281–290.
- [10] Ermira Daka and Gordon Fraser. 2014. A survey on unit testing practices and problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*. IEEE, 201–211.
- [11] Pedro Delgado-Pérez, Aurora Ramírez, Kevin J Valle-Gómez, Inmaculada Medina-Bulo, and José Raúl Romero. 2022. Interevo-tr: interactive evolutionary test generation with readability assessment. *IEEE Transactions on Software Engineering*, 49, 4, 2580–2596.
- [12] Pouria Derakhshanfar, Xavier Devroey, and Andy Zaidman. 2022. Basic block coverage for search-based unit testing and crash reproduction. *Empirical Software Engineering*, 27, 7, 192.
- [13] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. 2007. The daikon system for dynamic detection of likely invariants. *Science of computer programming*, 69, 1-3, 35–45.
- [14] Zhaopeng Feng, Yan Zhang, Hao Li, Wenqiang Liu, Jun Lang, Yang Feng, Jian Wu, and Zuozhu Liu. 2024. Improving llm-based machine translation with systematic self-correction. *arXiv e-prints*, arXiv–2402.
- [15] Gordon Fraser and Andreas Zeller. 2010. Mutation-driven generation of unit tests and oracles. In *Proceedings of the 19th international symposium on Software testing and analysis*, 147–158.
- [16] Sepideh Kashefi Gargari and Mohammad Reza Keyvanpour. 2021. Sbst challenges from the perspective of the test techniques. In *2021 12th International Conference on Information and Knowledge Technology (IKT)*. IEEE, 119–123.
- [17] Siqi Gu, Chunrong Fang, Qunjun Zhang, Fangyuan Tian, Jianyi Zhou, and Zhenyu Chen. 2024. Improving llm-based unit test generation via template-based repair. *arXiv preprint arXiv:2408.03095*.
- [18] Mark Harman and Bryan F Jones. 2001. Search-based software engineering. *Information and software Technology*, 43, 14, 833–839.
- [19] Mark Harman and Phil McMinn. 2009. A theoretical and empirical study of search-based testing: local, global, and hybrid search. *IEEE Transactions on Software Engineering*, 36, 2, 226–247.
- [20] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*, 437–440.
- [21] Claus Klammer and Albin Kern. 2015. Writing unit tests: it's now or never! In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 1–4.
- [22] Lei Ma, Cyrille Artho, Cheng Zhang, Hiroyuki Sato, Johannes Gmeiner, and Rudolf Ramlar. 2015. Grt: program-analysis-guided random testing (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 212–223.
- [23] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 1–13.
- [24] Michael Olan. 2003. Unit testing: test early, test often. *Journal of Computing Sciences in Colleges*, 19, 2, 319–328.
- [25] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 75–84.
- [26] Per Runeson. 2006. A survey of unit testing practices. *IEEE software*, 23, 4, 22–29.
- [27] Kuniaki Saito, Kihyuk Sohn, Chen-Yu Lee, and Yoshitaka Ushiku. 2024. Unsupervised llm adaptation for question answering. *arXiv e-prints*, arXiv–2402.
- [28] Abdelilah Sakti, Gilles Pesant, and Yann-Gael Guéhéneuc. 2014. Instance generator and problem representation to improve object oriented code coverage. *IEEE Transactions on Software Engineering*, 41, 3, 294–313.
- [29] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2020. Unit test case generation with transformers and focal context. *arXiv preprint arXiv:2009.05617*.
- [30] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.
- [31] Zejun Wang, Kaibo Liu, Ge Li, and Zhi Jin. 2024. Hits: high-coverage llm-based unit test generation via method slicing. (2024). <https://arxiv.org/abs/2408.11324> arXiv: 2408.11324 [cs. SE].
- [32] Xusheng Xiao, Sihan Li, Tao Xie, and Nikolai Tillmann. 2013. Characteristic studies of loop problems for structural test generation via symbolic execution. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 246–256.
- [33] Zhuokui Xie, Yinghao Chen, Chen Zhi, Shuiguang Deng, and Jianwei Yin. 2023. Chatunitest: a chatgpt-based automated unit test generation tool. *arXiv e-prints*, arXiv–2305.
- [34] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. 2023. No more manual tests? evaluating and improving chatgpt for unit test generation. *arXiv preprint arXiv:2305.04207*.
- [35] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2019. The fuzzing book. (2019).
- [36] Hong Zhu, Patrick AV Hall, and John HR May. 1997. Software unit test coverage and adequacy. *Acm computing surveys (csur)*, 29, 4, 366–427.