LLM-Seeded Evolutionary Testing (LSET): Enhancing Search-Based Software Testing by Incorporating Complex Method Sequences

Chao Ran Erwin Li

LLM-Seeded Evolutionary Testing (LSET): Enhancing Search-Based Software Testing by Incorporating Complex Method Sequences

by

Chao Ran Erwin Li

to obtain the degree of Master of Science

at the Delft University of Technology,

to be defended publicly on Tuesday June 17, 2025 at 10:00 AM.

Student number:	5283728	
Project duration:	September, 2024 – June, 2025	
Thesis committee:	Chair:	Dr. A. Panichella, Faculty EEMCS, TU Delft
	Daily supervisor:	Dr. M.J.G. Olsthoorn, Faculty EEMCS, TU Delft
	Daily co-supervisor:	A. Deljouyi, Faculty EEMCS, TU Delft
	Committee Member:	Dr. S.E. Verwer, Faculty EEMCS, TU Delft

TU Delft

Abstract

Writing test cases is an important yet complex task. Search-Based Software Testing (SBST) is an automated test case generation technique that aims to help developers by creating high-coverage test cases. Despite its strengths, a major limitation of this technique is that it often struggles with generating test cases that contain complex method sequences, as they have no semantic understanding of which methods are related. The recent advancement of Large Language Models (LLMs) offers a potential solution due to their natural language capabilities and applicability to software engineering tasks. However, LLMs often end up with lower coverage than SBST when directly compared due to lacking the output diversity that is required in software testing. This opens an opportunity to combine the exploratory power of SBST, with the semantic understanding of LLMs to make the test case generation process more effective in terms of test coverage and test structure. This thesis investigates the combination of these methodologies with our hybrid approach, LLM-Seeded Evolutionary Testing (LSET), which uses LLMs to generate tests that contain complex method sequences, and introduces this structure into the SBST process by serving as the starting point (seeds) of the algorithm to be evolved further. We conducted an empirical evaluation on a benchmark consisting of 35 JavaScript classes and found a significant branch coverage increase on 19 and 14 classes when compared to SBST and LLM-only baselines. However, when taking the combination of final SBST and LLM test suites, this gap reduces to 1 out of 35 classes. Beyond coverage, we also found a positive effect on structure when compared to tests generated by SBST, as the structure provided by the LLM tests can be further evolved to reach deeper branches while maintaining readability.

Acknowledgements

This thesis marks the culmination of my five-year study at TU Delft. It was a fulfilling time that has allowed me to grow professionally and personally. I would like to give my sincere gratitude to the people who have supported me. First and foremost, I want to thank my supervisors Mitchell Olsthoorn and Amir Deljouyi, who met with me every week to discuss ideas and provide feedback. Without their guidance, this thesis would have not been half of what it is. I want to thank Annibale Panichella for serving as the thesis advisor and chair and giving invaluable feedback. Additionally, I want to thank Sicco Verwer for reading my work and being part of the thesis committee. Finally, I want to thank the people close to me. My parents, who have always supported me and serve as my inspiration, as well as my sister, who I can always talk and laugh with. Finally, I want to thank my friends, who I can always count on.

Chao Ran Erwin Li
June 2025

Contents

1 Introduction

Writing tests is a standard software engineering practice of paramount importance, as to assure the functionality of the code. Yet, developers tend to neglect this task due to its labor-intensive, time-consuming and tedious nature [6, 39]. To assist developers in the testing process, many automated test case generation techniques and tools have been developed. Among these approaches, Search-Based Software Testing (SBST) is one of the most popular and effective techniques for achieving high code coverage, and detecting new vulnerabilities [3, 4, 1].

Despite its strengths, SBST has significant limitations that hinder its effectiveness in generating certain types of test cases and contribute to its limited adoption in industry.

1.1. Problem Statement

A test case consists of three components: input data, a sequence of ordered method calls (test structure), and assertions [41]. SBST tools struggle to generate test cases that involve complex sequences of method interactions [22, 29]. Internally, SBST tools treat methods as abstract entities (e.g., method1, method2) without considering their semantic relationships, leading to methods being combined randomly. This limitation restricts their ability to simulate real-world usage patterns within the codebase. Hence, SBST tools may produce tests that achieve high code coverage but potentially fail to validate meaningful behaviors of the system under test, creating a gap between the coverage metrics and the actual quality of the tests. Moreover, the lack of realistic test structure can itself be a limiting factor for coverage, as semantically meaningful sequences of method calls are often necessary to reach deeper or more complex code paths.

```
it("Test 1 for 'ShoppingCart'", () => {
    // Test
    const shoppingCart = new
        ShoppingCart();
    const item =
        "=48=$Iwde7/RX(l3±3'~!5}S;0B\n&w}
    pk\t\"WlXX|%=u6@G+*2'fLc08TXwt;bE<:
        /MV^tUUJ@l$#xI1A~Wn(afzmRYc^F";
    const assignment = 264;
    const anon = null;
    const removeItemReturnValue = await
        shoppingCart.removeItem(item,
        assignment, anon);
    const shoppingCart1 = new
        ShoppingCart();

    // Assertions
    expect(removeItemReturnValue).to
    .equal(undefined);
});
```

(a) SBST Test (SynTest)

```
describe('ShoppingCart', () => {
    let cart;

    beforeEach(() => {
        cart = new ShoppingCart();
    });

    ...

    it('should remove items from the
        cart', () => {
        cart.addItem('Orange', 0.79, 3);
        cart.removeItem('Orange', 1);
        expect(cart.items.find(item =>
            item.item ===
            'Orange').quantity).toBe(2);
    });

    it('should clear all items from the
        cart', () => {
        cart.addItem('Book', 9.99);
        cart.addItem('Pen', 0.79, 5);
        cart.clearCart();
        expect(cart.items.length).toBe(0);
    });
```

(b) LLM Test (GPT 3.5)

Figure 1.1: Examples of test cases: (a) generated by SBST, and (b) generated by an LLM.

This problem is illustrated in Figure 1.1a, which showcases a test generated by SynTest for the ShoppingCart.js class. The ShoppingCart class is a simple artificial example containing multiple methods such as addItem, removeItem, and clearCart, where each item is defined by its name, price, and quantity.

The test invokes the removeItem method in isolation. While this test could technically be useful as a bad-weather case (testing unexpected or invalid input), it does not simulate a realistic interaction with the class. A realistic test would involve first adding items to the cart and then removing them, reflecting actual usage patterns. However, in the entire test suite generated by SynTest, no such case was generated. Hence, the lack of proper test structure, leads to the removeItem method not being tested properly. This omission demonstrates SBST's inability to account for the relationships between methods, even for a simple class. Furthermore, the test is difficult to interpret due to its lack of coherence and meaningful context. The randomly generated inputs, and isolated method invocation make it unclear what the intent behind the test is.

In other words, the test cases generated by SBST often fail to align with the purpose of testing itself, as they lack readability, are difficult to maintain, and do not correspond to any realistic testing scenario. If such a test fails, a developer may be unsure whether the failure stems from a problem in the software or the test itself. Therefore, these tests are rarely suitable for inclusion in a maintained test suite without substantial manual effort.

Recent advancements in Large Language Models (LLMs) offers a potential solution to these limitations. LLMs have demonstrated an ability to understand code semantics and generate more natural-looking and human-readable test cases [8, 43, 53]. This can be observed in Figure 1.1b, which displays two test cases generated by OpenAI's GPT-3.5 [1] for the previously mentioned ShoppingCart class. In contrast to the SBST-generated tests, the LLM-generated tests exhibit a logical sequence of operations that align with realistic usage patterns (i.e. adding items before removing them). Moreover, the structured and logical flow of the LLM tests makes their purpose and intent immediately obvious.

However, LLMs are not a silver bullet for automated test case generation, as they introduce their own set of challenges. LLMs typically achieve lower coverage than SBST [24, 40], due to struggling to generate the diverse output required for software testing [47]. Furthermore, the non-deterministic nature of LLMs can make them unreliable, and unstable [25]. Although SBST is also non-deterministic, its search

[1] https://platform.openai.com/docs/models/gpt-3-5

process generally produces some executable test cases. In contrast, LLMs sometimes generate tests that are empty, broken, or contain hallucinated, nonexistent code [11, 36].

This unpredictability raises a reasonable question: why not simply issue multiple queries to the LLM until a valid and meaningful test case is returned? Firstly, LLMs have significant computational cost [54], especially compared to SBST, limiting their scalability. Besides resource limitations this approach is also limited in effectiveness. LLMs are trained to generate the most statistically likely next token based on their training distribution, rather than to explore rare or edge-case scenarios. Consequently, if a specific test structure or branch condition is scarce in the training data, the model may consistently fail to generate it, regardless of how many times it is queried.

Hence, the combination of SBST and LLMs presents an opportunity to join the strengths of both approaches and cover each other's weaknesses. SBST's systematic exploration can ensure high coverage, while LLMs' natural-language capabilities can improve the quality and structure of test cases.

One of the first successful hybrid approaches combining SBST and LLMs is CodaMOSA [17], which combines the Codex model with the Pynguin framework. CodaMOSA runs the search-based algorithm until its coverage increases stall. Codex is queried when such a coverage plateau is hit to assist the search process. Building on this, TELPA managed to improve upon this approach by incorporating program analysis results and counter-examples into the prompt to guide the LLM towards tests that cover difficult branches. However, these two works focus mostly on coverage improvements of their approaches compared to the baselines, rather than how realistic and structured the test cases are. This singular focus, combined with the fact that both these works are in Python, a language LLMs seem to be biased towards possible due to over-representation in the training set [44], highlights a significant gap in existing research. To our knowledge, no prior hybrid approach has targeted JavaScript, despite it being one of the most popular programming languages in the world [37]. Our work aims to address this gap by not only focusing on coverage, but also on generating more realistic and structured test cases, particularly within JavaScript.

1.2. Research Aim

This thesis investigates the potential for integrating LLMs into the (SBST) process to enhance test generation for JavaScript programs. Specifically, we explore various prompt engineering strategies to guide LLMs in generating complex method sequences and their integration into existing SBST frameworks, examining whether this leads to improvements in both test structure and test coverage.

To answer this question, we introduce our hybrid approach LLM-Seeded Evolutionary Testing (LSET). LSET leverages LLMs to generate test cases containing rich structure, which will serve as the starting point (seeds) for the SBST algorithm, thereby introducing complex method sequences into the process which will then be further evolved.

To evaluate our approach, we conducted a large-scale empirical study across 35 JavaScript classes from real-world libraries, comparing LSET to standalone SBST and LLM approaches. Our results indicated that LSET, when taking into account the LLM test suites used to generate the seeds, managed to score significantly higher coverage over the standalone baselines of SBST and LLM in 19 and 14 classes out of 35 respectively. Furthermore, across all classes, LSET yielded an average coverage increase of 16.77 and 5.58 percentage points compared to standalone SBST and LLM respectively. However, when combining the final test suites of SBST and LLM, the gap closes and LSET is only statistically superior in one class while equal in others, with a very slight average coverage improvement (82.66% vs 82.44%).

However, coverage was not our only objective. Regarding structure, we found that LSET can improve the structure of SBST test cases, creating tests with logical method sequences that explore deeper branches. Therefore, despite covering the same objective, the tests generated by LSET can be more realistic and maintainable. However, high variance was observed between the test cases in the final test suite, as not every test fully utilized the structure provided by the LLM.

In conclusion, this thesis makes the following primary contributions:

- An empirical evaluation of different prompt engineering strategies to determine the most suitable approach for generating complex test structures.
- The development of a parser to translate LLM output into SynTest's internal encoding, enabling LLM integration into the SBST framework.
- An empirical evaluation of LSET's effectiveness in improving test structure and overall coverage compared to baseline approaches.
- An open-source implementation and replication package, including all data, scripts for plot generation, and documentation explaining design decisions [18].

1.3. Thesis Outline

The rest of the thesis is organized as follows. Chapter 2 provides the necessary background information and discusses relevant related work on SBST, LLM-based test generation and hybrid approaches Chapter 3 explains the design decisions for our proposed LSET approach, while Chapter 4 details the experimental setup used to evaluate it. Chapter 5 presents the results, comparing coverage and structure across strategies. Chapter 6 discusses our findings and limitations. Finally, Chapter 7 concludes the thesis.

2 Background and Related Work

In this chapter, we discuss the relevant background and related work required for understanding this thesis. We start by discussing SBST approaches, followed by detailing how LLMs have been applied to software testing. Finally, we discuss the existing work for hybrid approaches that combine SBST and LLMs.

2.1. Search-Based Software Testing

Manual software testing is a complex, tedious, and error-prone process that can take up to as much as 50% of a developer's time [15]. To alleviate developers of this burden, researchers have developed various techniques to automate the test case generation process. Examples include fuzzing, which generates (semi)-random inputs to explore a program's behavior and uncover vulnerabilities, and symbolic execution, which uses a constraint solver to systematically traverse execution paths by treating program variables as symbolic values rather than concrete inputs. Among these approaches, Search-Based Software Testing (SBST) has been particularly successful, by showcasing the ability to achieve high levels of coverage [7] and fault detection [35, 12].

SBST is an automated test case generation technique that frames test generation as a meta-heuristic optimization problem, that can be approached using search techniques, like evolutionary algorithms (EA). The current state-of-the-art algorithms use EA's to initialize a set of random solutions, known as the population, which subsequently undergoes an evolutionary process to construct the final test suite.

The most basic form of EA's consists of these stages:

- **Initialization**: A population of randomly generated test cases is created, each representing a candidate solution. These test cases are then evaluated using a fitness function, which measures their effectiveness in achieving test objectives.
- **Selection**: The best-performing solutions, as measured by the fitness function, are chosen to form the next generation. This mechanism mimics the process of natural selection.
- **Crossover & Mutation**: To introduce variation, crossover combines two test cases to produce an offspring that inherits characteristics from both parents, while mutation introduces small random modifications to individual test cases. This helps maintain diversity and prevents the algorithm from getting stuck in local optima.
- **Termination**: The process iterates over multiple generations until a stopping criterion is met, such as achieving sufficient coverage, or reaching the pre-defined time limit.

While these operators effectively explore a diverse set of paths, they often struggle with generating and maintaining specific sequences of method calls. This limitation arises because genetic operators do not inherently recognize or preserve meaningful sequences during test generation, leading to the loss or disruption of meaningful method interactions [49]. Consequently, SBST may be unable to thoroughly test certain classes whose behaviors depend on complex sequences of method invocations.

SBST tools have long used seed-based techniques to incorporate meaningful method sequences into automatic test generation. Seeding refers to the process of initializing the test population with preconstructed inputs [29]. Examples include reuse of previous solutions, seeding required constants, and optimizing the seeds to ensure every method is invoked [13]. These approaches help SBST produce more semantically coherent tests, but they largely depend on prior code artifacts. In contrast, our approach leverages LLMs to create complex method sequences, without requiring previous data.

2.2. Large Language Models for Test Case Generation

Large Language Models (LLMs) are deep learning models trained on extensive natural language and code corpora. Their impressive abilities have led to them being applied to multiple software engineering tasks, such as code completion [14], code generation [45], and code testing [34].

Users interact with LLMs via prompts which are textual inputs used to guide the model's output. The process of designing, refining, and optimizing these prompts to achieve specific and desired responses from LLMs is known as prompt engineering, with studies showing that high-quality prompts are essential for good results [25].

LLMs are particularly skilled at producing human-like, readable code and can infer meaningful relationships between methods. This makes them particularly suitable for generating complex sequences of method calls that reflect realistic usage patterns, precisely the type of test structure that SBST struggles with. For example, rather than invoking a method like removeItem() in isolation, an LLM-generated test would first initialize the relevant object, add items to a collection, and then attempt removing it, mimicking the way a developer might test such behavior.

Examples of LLM-based test generation tools are TestSpark [32], TestPilot [34], and AthenaTest [43]. TestSpark is an IntelliJ IDEA plugin for unit test generation, designed with a focus on ease-of-use and developer integration. To address the challenge of LLM test often being non-compiling, they propose a feedback loop that detects compilation errors and iteratively refines prompts until syntactically correct test code is produced or the maximum number of iterations is reached.

Similarly embracing a feedback-driven approach, TestPilot targets JavaScript and focuses on generating high-quality tests by leveraging additional context from documentation and usage examples. It explores the package's exported object graph and mines fenced code blocks and JSDoc comments to build metadata-informed prompts. These are sent to an LLM, and any failing tests are iteratively refined using execution feedback, enabling the model to self-correct and improve coverage.

Finally, AthenaTest is an LLM-based tool designed to generate unit tests directly from natural language function summaries found in source code comments. It prompts a language model (specifically Codex) with the target method and its accompanying comment to produce test cases. A key finding from its evaluation is that developers preferred the LLM-generated tests over those produced by EvoSuite, particularly in terms of readability and understandability. This reinforces the idea that LLMs are especially effective at generating well-structured, human-like tests that align closely with how developers write and reason about code.

This strength in producing readable and contextually appropriate tests highlights the complementary nature of LLMs to traditional approaches. However, while LLMs excel in generating human-like code, they often fall short in terms of structural coverage.

Recent comparative studies further underline the continued importance of search-based methods. Tang et al. [40] compare ChatGPT-generated suites with EvoSuite and show that, while LLM tests are more readable, EvoSuite still achieves substantially higher branch and mutation coverage on the same benchmarks. Likewise, Abdullin et al. [2] evaluates SBST, symbolic execution, and several LLM-based generators side-by-side. Across 50 Java projects, SBST consistently attains the best structural coverage, whereas LLM suites excel mainly in naturalness and naming quality. Together, these results suggest that, despite their usability advantages, LLMs alone cannot match SBST's thorough exploration of hard-to-reach branches.

2.3. Hybrid Approaches Combining SBST and LLMs

While LLMs excel at generating human-like test code, they often fall short in achieving high structural coverage. Conversely, SBST is well-suited for thorough systematic exploration of the code, but tends to produce unnatural test cases. This complementary relationship has led to a growing interest in hybrid approaches that aim to combine the strengths of both techniques.

One of the first successful methods is CodaMOSA [17], which integrates LLMs into the Pynguin framework. As querying LLMs is computationally expensive, CodaMOSA first runs the SBST algorithm, and only prompts the LLM if a coverage stall is detected. This approach is particularly effective when specific input values are required to satisfy branch predicates, such as a string denoting a project's version number. In a large-scale evaluation over 486 Python modules, CodaMOSA achieved statistically significantly higher coverage on 173 and 279 benchmarks, with an average coverage increase of 10% and 9%, compared to SBST and LLM-only baselines.

Building upon CodaMOSA, TELPA (TEst generation via LLM and Program Analysis) [52] introduces a more targeted hybrid approach to address hard-to-cover branches in automated test generation. TELPA enhances LLM-based test generation by integrating program analysis techniques to provide enhanced context and guidance to the language model. Specifically, it uses backward and forward method-invocation analysis to find relevant object constructions and influencing calls. Furthermore, it identifies ineffective tests from previous generations as counter-examples. These counter-examples are incorporated into the prompts provided to the LLM, guiding it to generate diverse and effective tests that can reach the hard-to-cover branches. In an evaluation across 27 open-source Python projects, TELPA demonstrated significant improvements in branch coverage, achieving an average increase of 31.39% over Pynguin and 22.22% over CodaMOSA.

While both CodaMOSA and TELPA demonstrate the potential of combining SBST with LLMs, they are limited to Python and primarily optimize for coverage. In contrast, our work explores this hybrid potential in JavaScript and additionally emphasizes the structural quality of generated test cases.

3 Approach

This chapter presents our approach LLM-Seeded Evolutionary Testing (LSET), which improves SBST by leveraging LLMs to incorporate test cases with complex method sequences into the evolutionary process. Figure 3.1 provides a high-level overview of the LSET pipeline, which is composed of two main stages:

1. **LLM Seed Generation**, where we applied prompt engineering techniques to consistently generate high-quality test cases with complex method sequences.
2. **Interation & Execution**, which integrates the parsed test cases into SynTest's evolutionary pipeline and evolves them to generate the final test suite

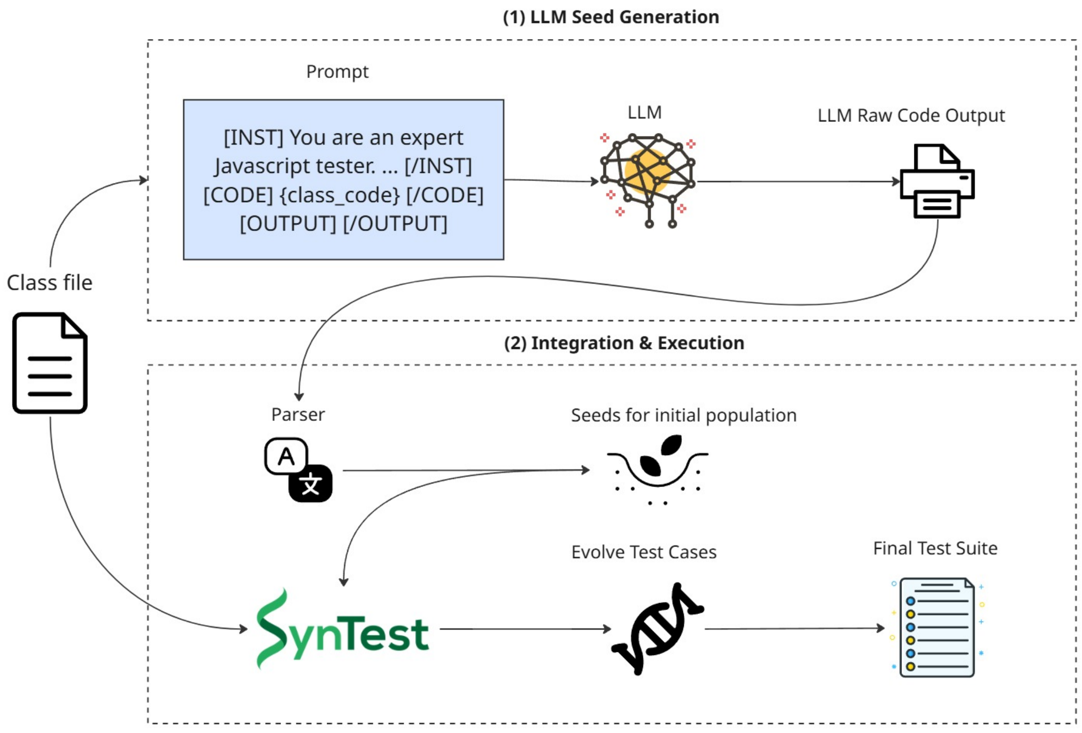

(1) LLM Seed Generation
Prompt
[INST] You are an expert Javascript tester. ... [/INST] [CODE] {class_code} [/CODE] [OUTPUT] [/OUTPUT]
LLM
LLM Raw Code Output
Class file
(2) Integration & Execution
Parser
Seeds for initial population
Evolve Test Cases
Final Test Suite
SynTest

Figure 3.1: LSET High Level Overview

The rest of this chapter will go into further detail behind the design decisions for each component of LSET.

3.1. Guiding LLM Behavior Through Prompt Engineering

To consistently generate effective tests with complex method sequences, we employ various prompt engineering techniques. Prior studies have shown that well-designed prompts are essential for producing reliable and relevant outputs [30, 25]. Given the importance of prompt design in influencing model behavior, we tested different techniques to determine which is most suitable for our purposes. The rest of this section first outlines the strategies we selected, followed by a breakdown of the final prompt designs.

3.1.1. Chosen Strategies

For this study, we focused on testing broadly different approaches rather than optimizing singular words in one prompt. Fine-tuning individual words can be difficult to evaluate consistently due to the non-deterministic nature of LLM outputs. Instead, our goal was to see how fundamentally different strategies affect the model's output.

In a literature study performed by Tony et al., they identify 15 techniques for code generation. They classify them into 5 main categories, namely: root-, refinement-based, decomposition-based, reasoning-based, and priming techniques.

Root techniques are the most basic forms of prompting, most likely to be used by average users. From this category, we chose the simplest technique, zero-shot, to serve as a baseline. This approach involves asking the model to generate a test, without any task-specific training or given examples in the prompt.

Refinement-based techniques involve asking the model to assess itself and improve upon its previous answer. We picked the Self-Refine strategy [20], which consists of 3 steps. Firstly, we ask the model to generate a test suite, without any special instructions. Next, our second prompt takes the output of step 1, and asks the LLM to reflect on it and provide feedback. Lastly, we take the test suite of step 1, and combine it with the feedback of step 2 in our final prompt to acquire the final refined test suite.

Decomposition-based techniques aim to simplify complex tasks by splitting them into more manageable steps. Our chosen technique from this category is Self-Planning [16]. This strategy consists of two phases, planning and implementation. In the planning phase, we first provide the LLM with multiple examples on how to produce a plan to generate a test suite. Subsequently, we ask it to produce a plan for a given class. In the implementation phase, we use the acquired plan, and use it as the instructions for the LLM to generate the final test suite.

Reasoning-based techniques seek to make full use of the logical reasoning capabilities of the model. The prompt asks the LLM to follow a logical path towards the final results, while communicating its thought process. We chose the most popular strategy of this category, Chain-of-Thought (CoT) prompting [50]. CoT encourages the LLM to produce a logical sequence of steps before reaching the final output, mimicking how humans tend to solve complex tasks. This technique consists of only one phase, in which we supply the LLM with an example set of steps on how to generate a test suite, before asking for the output.

Lastly, **priming techniques** involve asking the model to adopt a certain role for their response. For example, in our prompts we start the prompt with 'You are an expert JavaScript tester' to prime the model and specify the domain it is working in. For this pattern we decided to include it in all prompts, rather than creating a separate one, as it is a short and simple pattern that can easily be integrated into any prompt.

3.1.2. Prompt Breakdown

In this section we will explain how we devised the prompts and the guidelines we followed. To illustrate these design choices, we provide a full breakdown of the Zero-Shot prompt, as it is the simplest. For the other approaches, we will only highlight the sections that are unique to that particular strategy. All prompts can be found in their entirety in Appendix A.

For all strategies, we created a variant where we added additional instructions to make the model focus on generating complex method sequences. Our goal is to test whether these additional instructions affect the model's output positively, have no meaningful effect, or even degrade performance by making the prompt less clear or limiting the model's creative flexibility. We name these variants, the 'specific-variants' and call the base versions the 'generic-variants'. In all figures we highlight the additional instructions of the specific-variants in red.

Creating the prompts was an iterative process. We tested and refined each strategy through small-scale experiments using a simple artificial class and a small subset of the benchmark. The artificial class served as a 'best-case scenario', if a strategy failed to perform well here, it was unlikely to succeed on real-world code. For instance, we initially experimented with a Self-Planning style prompt that first asked the LLM to generate test scenarios, followed by a second prompt that implemented them. However, this approach often resulted in empty output, where the model would only generate a placeholder test with the scenario name but no actual implementation. Therefore, this version was discarded.

Zero-Shot Prompt

Figure 3.2 shows the prompt for the Zero-Shot strategy. In this prompt we follow multiple guidelines from prompt engineering research [19, 21]. We use pairs of tags (e.g. [OUTPUT] [/OUTPUT]) to create a standardized input-output structure. Furthermore, we adopt a persona ('You are an expert JavaScript tester), and define a goal with clear instructions ('Write test cases that are ...'). Lastly, we keep the instructions in the prompt concise to avoid overwhelming the model.

```
[INST]
You are an expert Javascript tester.
Your goal is to write test cases that are readable and understandable to other
developers while maximizing coverage of the class-under-test.
Include test cases with multiple method calls where relevant.
These involve sequences of method invocations on the same object, where one method's
    behavior or state affects others.
Use such tests when the 'class methods share state, have side effects, or depend on
    logical sequencing.
Avoid unnecessary complexity: if the methods are independent or do not benefit from
    sequencing, use standalone tests instead.
Place the final test suite between the [OUTPUT] and [/OUTPUT] tags.
The class-under-test is provided between the [CODE] and [/CODE] tags.
Do not return any text outside the [OUTPUT] and [/OUTPUT] tags.
[/INST]
[CODE] class_code [/CODE]
[OUTPUT] [/OUTPUT]
```

Figure 3.2: Zero-shot prompt, the red lines are included in the specific variants of the prompt

Chain-of-Thought Prompt

Figure 3.3 shows the extra reasoning-steps that are added in this strategy's prompt. These steps allow the model to reflect, and plan its approach step-by-step.

```
...

1. Carefully review the class-under-test provided between the [CODE] and [/CODE] tags.
2. Think and explain your reasoning step by step before writing the test cases:
   - Identify all methods and properties of the class.
   - Consider their expected behaviors.
   - Think through potential edge cases and boundary conditions.
   - Decide on the best test strategy to ensure full coverage.
   -Include test cases with multiple method calls where relevant.
    These involve sequences of method invocations on the same object, where one methods
         behavior or state affects others.
    -Use such tests when the class's methods share state, have side effects, or depend
        on logical sequencing.
    -Avoid unnecessary complexity: if the methods are independent or do not benefit
        from sequencing, use standalone tests instead.
3. Place only the final test code between the [OUTPUT] and [/OUTPUT] tags.
4. Do not include any text outside the [OUTPUT] and [/OUTPUT] tags.

...
```

Figure 3.3: CoT prompt snippet

Self-Refine Prompt

The Self-Refine strategy consists of three stages: an initial generation (identical to Zero-Shot), a reflection phase, and a final refinement step. As shown in Figure 3.4, the reflection prompt asks the model to critique its own test suite, while the refinement prompt guides it to improve the output based on that feedback.

```
--- Self-Refine Reflection ---
[INST]
Here is a test suite: self_refine_initial.
Please review the test suite and suggest any improvements or identify any missing edge
    cases or issues.
Provide feedback in terms of readability, coverage, and structure of the tests.
Specifically, check whether the test suite includes tests with multiple method calls
    where relevant.
These involve sequences of method invocations on the same object, where one method's
    behavior or state affects others

...

--- Self-Refine Refinement ---
[INST]
Here is the test suite to be improved   self_refine_initial.
Based on the following feedback, refine the test suite to include the suggested
    improvements:self_refine_reflection.
Ensure that the final test suite is comprehensive, readable, and well-structured,
    covering all the important scenarios for the class-under-test,including tests with
     multiple method calls where relevant

...
```

Figure 3.4: Snippets of the Self-Refine prompt stages. Self-Refine-initial omitted as it is identical to the Zero-Shot prompt

Self-Planning Prompt

Figure 3.5 shows the Self-Planning prompt, which is divided into two stages: planning and implementation. In the planning stage, the model is asked to create a structured test plan using a provided example as a template. The implementation stage then instructs the model to generate test cases based on the previously created plan.

```
--- Self-Planning Plan ---
[INST]
You are an expert JavaScript tester.
Review the provided class and create a structured testing plan like in the examples.
The plan should cover all essential scenarios and boundary conditions, including test
    cases with multiple method calls where relevant.
These involve sequences of method invocations on the same object, where one method's
    behavior or state affects others
...

[EXAMPLE]
class Calculator {
  add(a, b) {{ return a + b; }
  subtract(a, b) { return a - b; }
  divide(a, b) { return a / b; }
}}

Plan:
1. Test addition with positive and negative numbers.
2. Test subtraction with zero, positive, and negative numbers.
3. Test division by non-zero values and handle division by zero.
4. Test large and small input values for potential overflow.
[/EXAMPLE]
...

--- Self-Planning Implementation ---
[INST]
Here is a detailed plan: self_planning_plan.
Based on this plan, implement JavaScript test cases that maximizes coverage and are
    easy to understand for other developers.
Ensure that the test cases include tests with multiple method calls where relevant, as
    per the plan.

...
```

Figure 3.5: Snippets of the Self-Planning prompts

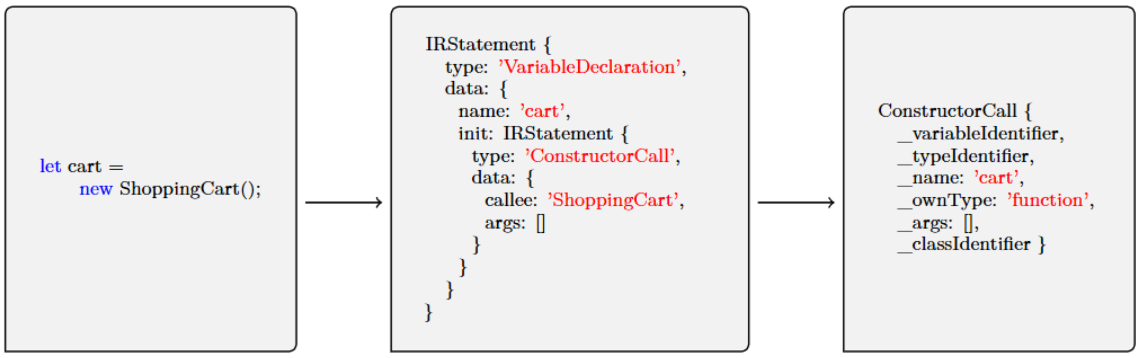
3.2. Integrating Complex Method Sequences into SBST

In this section, we describe the design decisions behind integrating meaningful test structure into the SBST pipeline. Firstly, we introduce our Intermediate Representation (IR) that serves as a translation layer between LLM output and the SBST's inner encoding, allowing for greater flexibility through this decoupling. Next, we explore different potential approaches and locations for incorporating the structure. Finally, we justify our choice of using seeding as the most suitable approach.

3.2.1. Bridging LLM Output and SBST Encoding via an Intermediate Representation

As LLM output is not directly usable by the SBST framework, we designed a parser to bridge this gap and translate test cases between encodings. In order to maintain flexibility, we first translate into an abstraction layer, the Intermediate Representation (IR), which stores all necessary semantic information. By decoupling the translation process from the integration mechanism, it enables us to support any potential integration method, rather than being forced into a single approach.

While it is technically possible to translate directly into SynTest's internal encoding, this would result in tightly coupled logic that is harder to maintain and extend. The separation of concerns provided by the



```
let cart =
    new ShoppingCart();
```

```
IRStatement {
  type: 'VariableDeclaration',
  data: {
    name: 'cart',
    init: IRStatement {
      type: 'ConstructorCall',
      data: {
        callee: 'ShoppingCart',
        args: []
      }
    }
  }
}
```

```
ConstructorCall {
  _variableIdentifier,
  _typeIdentifier,
  _name: 'cart',
  _ownType: 'function',
  _args: [],
  _classIdentifier }
```

(a) Raw Code **(b)** Intermediate Representation (IR) **(c)** Syntest Encoding

Figure 3.6: An overview of the parsing process from raw code to IR to SynTest encoding

IR improves modularity, simplifies transformations (such as injecting setup logic), and allows for easier debugging of individual pipeline components.

Figure 3.6 shows an example of how a single statement is transformed in each stage of the parsing pipeline. The first box contains a JavaScript line that declares a variable cart and initializes it with a new instance of the ShoppingCart class. The second stage involves parsing into the IR, where we extract and store all relevant semantic information. In this case it is the type of the variables (VariableDeclaration & ConstructorCall, name (cart), callee (ShoppingCart), and their respective arguments. In the final stage, the IR is matched to the corresponding SynTest statement and all data is translated accordingly.

The IR allows us to store and represent all relevant types of code, even those not currently supported by SynTest. The in between layer allows us to easily modify the test cases to make them compatible with SynTest. For instance, SynTest does not currently have a beforeEach statement, but this construct is common in the output of the LLM. The IR allows us to store the content and inject it into every test case. Similarly, SynTest does not support chained method calls like list.insert(1).insert(2).insert(3), so we must split them into separate statements to ensure each call is parsed individually.

By preserving more information than is currently needed, the IR makes the system more robust to change. As the test generation framework evolves to support additional constructs, IR statements can be reinterpreted accordingly, without requiring significant changes to the IR structure itself.

In short, this abstraction makes our design approach-agnostic: it decouples representation from integration, allowing us to explore a variety of strategies for incorporating LLM-generated test logic into the SBST process. We discuss these possibilities in the following section.

3.2.2. Exploring Integration Possibilities

Once test cases have been translated into the IR, the next step is to integrate them into the evolutionary process. In this section, we analyze the different possibilities in which LLM-generated structure can be incorporated into the SBST pipeline, along with the benefits and trade-offs accompanying each approach.

Seeding with Entire Test Cases

The most straightforward approach is to seed the initial population with entire LLM-generated test cases. Seeding refers to using prior knowledge to assist in solving a problem [29]. In our context, we apply this concept by populating (part of) the initial population with test cases generated by the LLM, allowing the evolutionary algorithm to begin from a more informed position rather than a completely random set of solutions.

Prior studies have demonstrated that seeding with manually created or previously generated test cases can enhance efficiency and coverage [13, 29]. An additional benefit of seeding is that it only requires querying the LLM once before the evolutionary process begins. Since generating output from an LLM is very time-consuming, we want to avoid repeatedly invoking the LLM in the search-loop. Lastly, with seeding we can fully utilize any secondary benefits the LLM provides, such as realistic input values for and complex object initialization that would otherwise be difficult for SBST to generate. However, a limitation of this approach is that there is no guarantee that the structures introduced at initialization are preserved over the evolutionary process.

Extracting and Preserving Complex Method Sequences As Linkage-Structures
A more complicated approach involves first analyzing the LLM-generated test cases to discover recurring method combinations and treating these as linked building blocks. This concept is based off linkage learning in evolutionary algorithms, where understanding variable dependencies improves search effectiveness by preserving beneficial combinations during evolution [48].

Traditional linkage learning empirically discovers interdependent variables by analyzing which combinations consistently appear in high-fitness solutions during evolution [48]. It is difficult to apply this in our context due to the vast search space, making it unlikely for the search-process to naturally produce complex structures. Furthermore, since SBST lacks access to semantic information, it would require a high-number of generations to identify relationships that are obvious from a programming perspective.

By employing an LLM, we can leverage it's semantic understanding of method dependencies to learn the linkage-structures rather than discovering them empirically. After identifying common method sequence patterns from LLM output, these structures can be utilized at various stages of the SBST process, such as:

- **Crossover**: Design recombination operators that preserve LLM-derived linkages
- **Initialization**: Construct the initial population using identified method combinations as building blocks,
- **Mutation**: Bias mutation operators to insert, modify, or replace entire linked sequences rather than individual method calls.

A comprehensive evaluation could examine all possible combinations of these three integration points to determine which stages benefit most from linkage-aware operations and whether combining multiple stages provides cumulative contributions or introduces competing constraints.

The main benefit of this approach is that complex method sequences are preserved over the entire evolutionary process. However, a notable drawback is the additional computational overhead that is required for linkage identification and extraction. Moreover, the learned patterns may over-constrain the search space, leading to reduced population diversity or premature convergence.

Rationale for Seeding as the Final Choice
While multiple integration strategies were explored conceptually, we ultimately chose to pursue seeding as the integration method in this work. The primary reason for this decision was practical, as seeding is significantly simpler to implement and integrates naturally with the existing SBST pipeline without requiring changes to the internal search operators. In contrast, a linkage-based approach would have required developing a tool to extract recurring method sequences from LLM-generated tests, followed by the design and implementation of new crossover and mutation operators to preserve these structures during evolution. Given the time constraints, pursuing both approaches was not feasible. Seeding offered a simple integration method that still effectively incorporated the LLM-generated structure, making it the most viable option within the scope of this thesis.

4 Study Design

This chapter outlines the methodology used to address the research questions presented in this thesis. The evaluation is structured around two distinct experimental setups: one focused on investigating prompt engineering strategies for generating complex test cases, and the other dedicated to evaluating the integration of LLMs into the SBST process.

4.1. Research Questions

The following research questions guide this thesis and its experimental evaluation.

- **RQ1** Which prompt engineering strategies are most effective for generating test cases with complex method sequences?
- **RQ2** How effective is the IR parser at preserving the LLM-generated test cases?
- **RQ3** How effective are different test generation approaches w.r.t. branch coverage and complex structure?
  - **RQ3.1** How effective are the baseline approaches, SynTest and GPT-4o-mini, w.r.t. branch coverage and generating complex test cases?
  - **RQ3.2** How effective is the LSET approach w.r.t. branch coverage and generating complex test cases?
  - **RQ3.3** To what extent do LLM-generated tests resemble existing human-written tests, and what does this reveal about potential data contamination?
- **RQ4** How do different configurations of the LSET approach impact test generation performance w.r.t. branch coverage?
  - **RQ4.1** How does the number of LLM-generated seed tests affect the performance of LSET?
  - **RQ4.2** What is the impact of invoking the LLM during the evolutionary search budget compared to prompting it beforehand, outside the search time?

4.2. Benchmark

This section outlines how we formed the benchmark that was used in all experiments. Our goal is to test the effectiveness of our prompt templates on creating complex method sequences in test cases. Therefore, the classes in the benchmark should be complicated enough to require this advanced test setup. Hence, we chose two criteria for a file to be included in the benchmark.

1. The number of functions should be >2
2. The class needs to have high-cohesion as measured by LCOM

The first requirement is included, as when there are only 2 functions, there are only 4 total combinations of methods. Randomly combining methods will most likely net you the correct combination, making usage of LLMs redundant. The second requirement uses the Lack of Cohesion Metric (LCOM) [9] as a heuristic to find classes with interdependent method sequences. Cohesion in software refers to the degree of interdependence among different methods. LCOM measures this by checking how many methods access the same instance variables. If most methods operate on different data, it results in low cohesion. If a class has no cohesion, the methods should not be called in a specific sequence as they are unrelated. Therefore, classes lacking in cohesion are unsuitable for our purposes.

We utilize the SynTest-JavaScript-Benchmark [38] as our starting point. This benchmark is one of the few existing benchmarks specifically designed for unit-level test case generation in JavaScript. The current version consists of 99 files from five widely used open-source JavaScript libraries: Commander.js, Express, JavaScript Algorithms, Lodash, and Moment.js. After applying our filtering criteria, we are left with 38 remaining files: 2 from Express, 10 from Moment and 26 from JavaScript Algorithms.

Since these libraries are very popular, they are (almost) guaranteed to be in the training data of the LLM model. While data leakage is a significant concern, the construction of a new benchmark is a time-consuming and labor-intensive process. Therefore, we have opted to limit our scope to the SynTest benchmark. This means our results are most likely biased in favor of the LLM, which we will take into account when interpreting our results and making any claims,

4.3. Evaluating Different Prompt Engineering Strategies

To determine which prompt engineering strategy most effectively guides the LLM toward generating high-quality test cases, we conducted an empirical evaluation. This section outlines the criteria and protocol used to assess each strategy's performance.

4.3.1. Evaluation Criteria

To evaluate the results of the different prompt strategies, we defined the following evaluation criteria:

- **Objective Metrics:**
  - Branch Coverage
  - Time Taken
  - Output Stability
  - Consecutive & Total Method Calls
- **Subjective Metrics:**
  - How realistic is the input data?
  - Does it successfully combine related methods?
  - How logical are the test cases?
  - Does it find edge cases?

The first metric is coverage, as it is a widely-used standard used to evaluate test case quality [27]. In software testing, code coverage refers to the degree to which the source code of the class-under-test is executed when running the test suite. We also measure the time it takes per strategy to produce their output, to gauge the practical differences.

The next measurement is output stability, which we define as the number of times the LLM returns anything that is compilable. If the test suite is empty, or none of the tests are compilable, the result is deemed invalid. Note, if a single test case has a syntax error, the entire test suite becomes incorrect. As this experiment is on a smaller scale, we manually corrected minor errors or removed the invalid test case. We still deem these test suites as correct, as it is possible to automate this process. Since this metric only requires a test to be syntactically correct, it may classify test suites as valid even if they are meaningless in terms of coverage. We consider this a minor issue, since our framework contains coverage as a separate criterion. Therefore, even if a test suite is nonsensical, this is taken into account in a different part of our evaluation.

Since our goal is to identify the most effective prompting strategy for capturing meaningful method structure, we introduce a metric based on method calls. Specifically, we measure both consecutive and total method calls. A method call is considered consecutive when it is invoked on the same object as the previous method call. The goal of this metric is to capture the extent to which test cases are able to construct complex method sequences. In addition to consecutive calls, we also count the total number of method calls in a test case as a proxy for its overall complexity. For both of these measurements, we report two statistics: the average number of calls per test suite and the maximum number of calls observed within a single test case.

Moving on from the objective- to the subjective metrics, we rate each test suite on a 0–2 scale for each subjective criterion. A score of 0 indicates a strong result: "the test suite has no major issues and would require minimal changes." A score of 1 reflects an acceptable but imperfect result: "the suite has issues, but is still usable." A score of 2 represents a poor result: "the output is either missing or not useful." These criteria were chosen with the overarching theme being: *Would I (a competent developer) write these tests?*

We measure all objective metrics via automated scripts, while the subjective metrics require manual analysis. We sample a random subset of the results for this, as the sheer number of test suites is too overwhelming to analyze every single one.

4.3.2. Experimental Protocol

Each experiment was repeated across 10 independent runs per strategy to account for variability in LLM outputs. We used the GPT-3.5-turbo model via the OpenAI API, with default parameters, including a temperature of 1.0 and no system prompt. Coverage was measured using Jest and the other metrics were measured using a custom AST parser.

For this research question, we will only use 20 of the 35 classes in the benchmark, since the analysis includes a manual component. This analysis focuses on comparing overall performance across strategies, while a more thorough investigation of class-level performance is reserved for RQ3, where we compare the LLMs performance to other approaches.

At the time of this experiment, we used GPT-3.5-turbo due to its accessibility, and low cost. While GPT-4 variants were available, they were assumed to be impractical due to higher pricing. Only after completing this phase of the study did we identify GPT-4o-mini as a more affordable alternative. As such, later experiments use GPT-4o-mini for improved model quality at a lower cost.

4.4. Evaluating the Integration of LLMs into SBST

This section presents the methodology used to address **RQ2** through **RQ4**, which investigate the integration of LLMs into the SBST process via the LSET approach.

Due to the JavaScript's complexity and the wide variety of possible syntax patterns, achieving full parser coverage would require a significant amount of implementation time. Since this thesis focuses on integration and experimental evaluation, we prioritized the most common patterns found in test cases. Once the parser reached a stable state for typical cases, we proceeded with experiments, leaving full parser coverage for future work.

To take into account the imperfections of the parser we introduce the following research question:

- **RQ2:** How effective is the IR parser at preserving the LLM-generated test cases?

In addition to RQ2, we explore the test generation capabilities of different approaches and configurations:

- **RQ3:** the effectiveness of different test generation strategies in producing complex method sequences, and coverage
- **RQ4:** the impact of different LSET configurations on test generation performance.

We use branch coverage as the primary metric for evaluating test generation quality. Coverage serves as a practical and widely accepted proxy for test effectiveness, as potential bugs can only be detected if the code is run. Furthermore, we will use the consecutive method calls metric, described in the previous section. Lastly, we conduct a manual analysis of specific classes to understand why certain approaches achieve higher coverage over others.

4.4.1. Evaluation Configurations

Table 4.1 summarizes the configurations evaluated for each research question.

Table 4.1: Overview of evaluation configurations per research question.

RQ	Focus	Configurations Compared
RQ2	Parser Preservation	LLM vs Parsed-LLM
RQ3.1	Baseline Effectiveness	SynTest, LLM, SynTest + LLM
RQ3.2	LSET Effectiveness	LSET, LSET + LLM (vs. baselines)
RQ4.1	Seed Size	LSET-10, LSET-25, LSET-50 (+ LLM variants)
RQ4.2	Search-Time Prompting	Each configuration with prompting *before* vs. *during* search

RQ2: Parser Preservation — We compare test coverage between the test suites generated by the LLM with their parsed counterparts to assess the capabilities of the parser. For this evaluation, we call the LLM exactly once per class in each run.

RQ3.1: Baseline Effectiveness — We evaluate the standalone performance of SynTest and GPT-4o-mini to establish baseline coverage performance for generating complex test cases. Additionally, we evaluate the union of these two strategies to assess whether they cover different parts of the class under test, and to determine whether the LSET approach can achieve greater coverage than this straightforward combination.

To fairly compare the LLM baseline against the LSET approach, we match the number of LLM-generated test cases used. In LSET, LLM prompting is repeated as needed until the specified number of seed tests (e.g., 10, 25, or 50) is reached. This is necessary because a single LLM invocation rarely returns an exact number of usable test cases. For fairness, we apply the same rule to the standalone LLM baseline. Meaning, instead of a single returned test suite, we repeatedly call the LLM until the cumulative number of test cases matches the corresponding LSET configuration. Since LSET-25 was the best-performing variant, the baseline LLM is run until it generates approximately 25 tests per class.

RQ3.2: LSET Effectiveness — We compare LSET against each of the baselines, using its best configuration. This includes prompting outside search-time, rather than within. This approach yielded slightly better performance in our experiments, and more closely reflects practical usage, as a minor additional setup is acceptable for superior results.

We also evaluate LSET + LLM against the baselines. This union consists of both the evolved test suite and all LLM-generated test suites which were generated in the process. This configuraiton represents the full potential coverage produced by the LSET approach, as the LLM-generated tests are required for seeding and can therefore be reused in the final test suite at no additional cost.

RQ3.3: Quantifying Similarity of LLM-Generated Tests to Manual Tests — In addition to evaluating coverage and complexity, we also assess whether LLM-generated tests resemble existing human-written ones, to investigate whether LLMs are copying existing tests (memorization) or are producing novel ones (generalization).

To conduct this analysis, we adopt the approach introduced by [17], which calculates the similarity score of each test case relative to code present in the project but excluded from the prompt. Similarity is measured using edit distance [23]. We employ substring-level matching with a sliding window, as prior research indicates that this method yields more accurate similarity scores, albeit at increased computational cost [28].

Classes without existing manually written test suites are excluded. For the classes in our benchmark, all classes from the JavaScript Algorithms library include test files, while only three classes from Moment do, and none from Express.

A potential way to test for generalization is to check whether LLM-generated test suites cover more branches than the manually written ones. However, we omit this comparison because the manual test suites in JavaScript Algorithms already achieve maximum branch coverage, leaving no uncovered branches to discover. While this analysis would still be possible for the three Moment classes, we consider the sample size too small to draw meaningful conclusions.

RQ4.1: Seed Size Impact — We investigate the effect of using 10, 25, or 50 LLM-generated seed tests to fill the initial population, both with and without including the LLM-generated tests in the final suite:

- LSET-10 vs LSET-25 vs LSET-50
- LSET-10 + LLM vs LSET-25 + LLM vs LSET-50 + LLM

RQ4.2: Search-Time Prompting — We measure the impact of when the LLM is prompted: either before the SBST search begins (OUT) or during the search process (IN).

Each configuration is tested under both conditions:

- LSET-10-OUT vs LSET-10-IN
- LSET-25-OUT vs LSET-25-IN
- LSET-50-OUT vs LSET-50-IN
- LSET-10-OUT + LLM vs LSET-10-IN + LLM
- LSET-25-OUT + LLM vs LSET-25-IN + LLM
- LSET-50-OUT + LLM vs LSET-50-IN + LLM

4.4.2. Parameter Settings

All SBST experiments were conducted using the state-of-the-art DynaMOSA algorithm [26], as implemented in the SynTest framework. Previous work has shown that, while parameter tuning impacts performace, default parameters provide acceptable results [5]. Hence, we use the default parameters provided by SynTest for the DynaMOSA algorithm [1] for all experiments, with the most notable settings being a population size of 50 and a fixed search-time budget of 180 seconds per run.

For seeding we choose 10, 25, or 50 seeds, filling the remaining slots with randomly generated solutions. We query the LLM until it returns the desired amount of tests. To improve efficiency, we use the same prompt for each query and issue multiple calls in parallel. The number of parallel invocations is determined dynamically based on the number of remaining required test cases. Since the number of tests returned per call varies across classes, this strategy may lead to over- or under-generation. Users can set the number of parallel calls to trade off between efficiency and the risk of generating unused tests.

All LLM test generation was performed using the GPT-4o-mini model with its default configuration, and no system prompt. The most relevant parameter in this context is the temperature, which was kept at 1.0.

4.4.3. Experimental Protocol

To mitigate the inherent stochasticity of both evolutionary algorithms and LLMs, we repeated the experiment 10 times for each configuration with different random seeds.

We performed the experiment on a system with 2 AMD EPYC 7H12 (64 core, 3293.082 MHz, 256 threads) processors with 512 GB of RAM. We ran 64 cores in parallel for the standalone SynTest configuration, and only 5 for all configurations involving LLMs due to OpenAI rate limits.

To determine whether one configuration is statistically superior to another, we utilize the Wilcoxon signed-rank test [51] with a significance level of $p = 0.05$. Furthermore, we apply the Vargha-Daleney $\hat{A}_{12}$ statistic [46] to measure the effect size of the result. This statistic measures the probability that a randomly selected result from one configuration will outperform a randomly selected result from another.

[1] https://github.com/SynTest-framework/SynTest-framework/blob/main/tools/base-language/lib/presets/DynaMOSAPreset.ts

5 Results

5.1. RQ1: What prompt engineering techniques are most effective for generating complex method sequences in test cases using LLMs?

This section discusses the effectiveness of various prompt engineering strategies in generating complex method sequences. The evaluation utilizes both objective and subjective metrics to assess the strategies. While this analysis focuses on overall performance across strategies, a more detailed investigation of class-level behavior is reserved for RQ3, where we compare LLM-generated tests to those produced by other approaches.

5.1.1. Objective Metrics

Branch Coverage

Figure 5.1 presents the median branch coverage achieved by each prompt engineering strategy across the benchmark classes, averaged over 10 runs. The boxplot shows that Zero-Shot-specific has the highest median with 68.09%, while Chain-of-Thought-specific has the lowest with 53.33%. This indicates that the additional complexity introduced with the Chain-of-Thought prompt does not translate into more coverage, at least within the tested contexts.

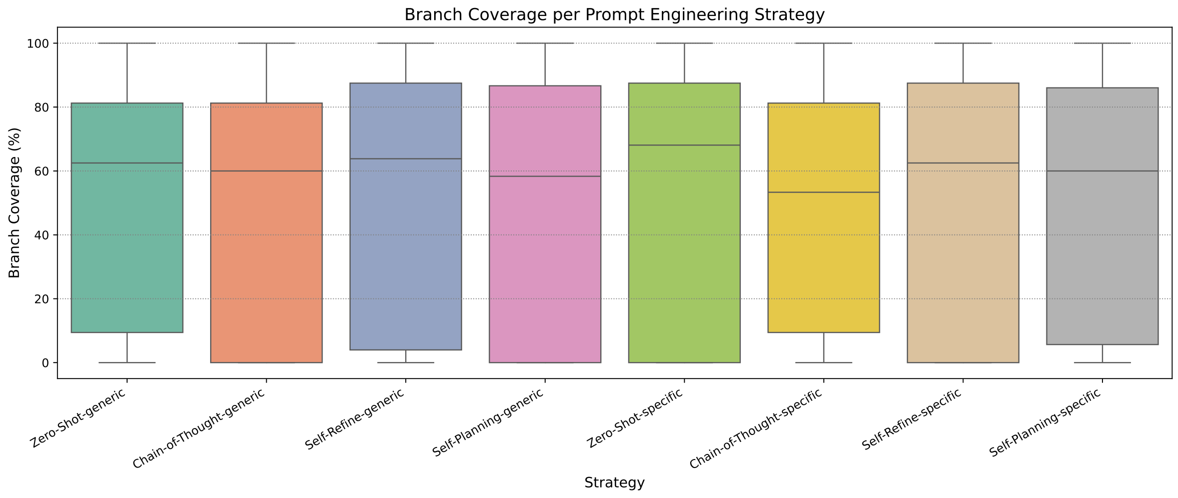


Figure 5.1: Boxplot displaying median branch coverage per class for each strategy over 10 runs

Output Stability

Stability statistics are shown in table 5.1. There was great variance in the number of invalid test suites between the approaches. Zero-Shot-specific was the best with faultless stability across all runs, while Self-Planning-generic struggled considerably with 19 invalid test suites out of 200. Table 5.2, displays which classes caused these errors, with the top 3 classes causing 30 out of 44 invalid test suites. These results suggest that instability is often tied to specific challenging classes, rather than reflecting a general weakness across all test cases.

Notably, the specific variants have equal or more stable performance, compared to their generic counterparts for every strategy. This indicates that adding this additional instruction is beneficial in this area.

Table 5.1: Invalid Test Suites by Strategy out of 200

Strategy	# Invalid
Zero-Shot-specific	0
Self-Refine-specific	1
CoT-specific	1
Self-Refine-generic	1
Zero-Shot-generic	3
Self-Planning-specific	8
CoT-generic	11
Self-Planning-generic	19

Table 5.2: Distribution of classes responsible for generating an invalid test suite

Class	# Invalid
compare.js	12
bfTravellingSalesman.js	9
bubble.js	9
from-anything.js	4
min-max.js	4
stronglyConnectedComponents.js	4
bitsToFloat.js	1
KnapsackItem.js	1

LLM Prompt Time

We analyzed the time taken by each strategy to produce its output, with the results displayed in Table 5.3. As expected, Self-Refine takes the longest, followed by Self-Planning, as they consist of multiple stages. Notably, the specific variants all take lower average time than their generic counterparts, indicating that the additional instructions can help narrow down the space and accelerate the LLM's computation.

Table 5.3: Comparison of Generic and Specific Strategies for Average Timing Performance

Strategy	Mean	Min	Max	Std
Generic Strategies				
Zero-Shot-generic	5.18	2.40	17.54	3.27
CoT-generic	5.96	2.67	12.18	2.60
Self-Refine-generic	16.97	11.74	23.06	2.97
Self-Planning-generic	9.92	6.19	17.31	2.82
Specific Strategies				
Zero-Shot-specific	4.87	3.43	9.42	1.30
CoT-specific	4.69	3.14	6.25	0.99
Self-Refine-specific	13.75	10.21	25.71	3.45
Self-Planning-specific	8.01	5.31	11.54	1.63

Complexity Metrics

Table 5.4 presents the final set of objective metrics: the complexity of the generated test cases, measured in terms of both consecutive and total method calls.

The most notable result is the strong performance of the **Self-Refine-specific** strategy, which outperforms all other strategies across all four metrics. It has the highest average and maximum number of both consecutive and total method calls, indicating that it produces not only more complex test cases, but also more structured sequences of interdependent method calls.

Comparing specific and generic variants more broadly, we observe that Self-Refine-specific and Self-Planning-specific outperform their generic counterparts in every complexity metric, suggesting that more targeted prompting leads to richer method sequences. Interestingly, the opposite is true for the Zero-Shot and Chain-of-Thought strategies, where the generic versions marginally outperform the specific ones.

Table 5.4: Comparison of Specific and Generic Strategies for Method Structure Complexity Metrics

Strategy	Average Consecutive Calls			Average Method Calls			Max Consecutive Calls			Max Total Method Calls		
	Mean	Median	Std	Mean	Median	Std	Mean	Median	Std	Mean	Median	Std
Generic Variants												
Zero-Shot-generic	1.20	1.00	1.39	2.24	1.58	2.52	1.83	1.00	1.92	3.23	2.00	3.19
CoT-generic	1.09	1.00	1.13	2.22	1.73	2.18	1.80	1.00	1.91	3.30	3.00	2.95
Self-Refine-generic	1.07	1.00	0.92	1.90	1.63	1.37	1.93	1.00	1.75	3.47	3.00	2.75
Self-Planning-generic	0.92	0.78	1.06	1.77	1.57	1.62	1.70	1.00	1.78	3.13	3.00	2.66
Specific Variants												
Zero-Shot-specific	1.12	1.00	0.88	2.07	1.67	1.61	1.70	1.00	1.45	3.17	3.00	2.49
CoT-specific	0.99	1.00	0.87	2.05	1.60	1.41	1.52	1.00	1.44	3.16	2.00	2.47
Self-Refine-specific	1.27	1.14	1.00	2.38	1.93	1.73	2.35	2.00	1.75	4.25	4.00	3.06
Self-Planning-specific	1.01	1.00	0.97	2.05	1.86	1.51	2.06	2.00	1.86	4.05	4.00	3.04

5.1.2. Subjective Metrics

Due to the high number of invalid test suites produced by Chain-of-Thought-generic and Self-Planning-generic, and the absence of other noteworthy qualities, we excluded these strategies from the manual analysis, as they were clearly not viable candidates for final selection.

Strategy	Understandability	Complexity	Realistic Input Data	Edge Cases
Zero-Shot-generic	0.2	0.2	0.0	1.6
Zero-Shot-specifc	0.0	0.2	0.2	1.4
CoT-specific	0.2	0.2	0.0	1.4
Self-Refine-generic	0.4	0.2	0.0	0.8
Self-Refine-specific	0.4	0.0	0.2	0.6
Self-Planning-specific	0.4	0.0	0.0	1.2

Table 5.5: Average subjective evaluation scores per strategy (lower is better)

With the exclusion of these two strategies, we end up with a full dataset of 1.200 generated test suites (20 classes × 6 strategies × 10 runs). We manually analyzed a subset of 30 for subjective evaluation. This subset was randomly sampled across strategies and runs. Although the sample represents only a small fraction of the total, preliminary inspection revealed minimal variation in the characteristics under review. Given the time-intensive nature of manual analysis and the consistency observed in early samples, we limited our review to this initial subset.

Table 5.5 shows the averaged scores for each metric. The results across the evaluated strategies are relatively close, suggesting that the LLM is generally capable of producing logical, and realistic test cases with complex structure regardless of prompt strategy. Interestingly, there did not seem to be a notable difference between generic- and specific variants in combining method sequences as all strategies managed to accomplish this in most cases. This indicates that the LLM can accomplish this without additional instructions.

The most notable variation appears in the edge case metric. Here, Self-Refine achieves better scores than other approaches. This suggest that the reflection phase of the Self-Refine approach is beneficial in discovering missing scenarios.

5.1.3. Final Choice

To determine the most effective prompting strategy for integration into our SBST workflow, we applied a process of elimination based on the acquired results.

Zero-Shot-generic and Self-Refine-generic performed reasonably well, but were consistently outperformed by their specifics variant across most metrics and were therefore excluded. Chain-of-Thought-generic and Self-Planning-generic were eliminated early due to their instability, and lack of other outstanding qualities. Chain-of-Thought-specific was also excluded, as it had the lowest median branch coverage (53.33%) and showed no standout performance on any metric.

This left two remaing candidates: Zero-Shot-specific and Self-Refine-specific. Between these two, Zero-shot-specific achieved the highest branch coverage (68.09%), but underperformed in terms of edge case coverage and complexity metrics. Conversely, Self-refine variants demonstrated stronger performance on edge cases and test case complexity, but with lower coverage.

Ultimately, we selected Self-Refine-specific as the final strategy. As despite it performing below Zero-Shot in generating coverage, it aligns more with our overall goal of generating method sequences that SBST tools struggle with. Furthermore, The lack of coverage can be mended by combining it with SBST tools

One trade-off of this choice is execution time. Self-Refine involves three prompting stages making it the most time-consuming strategy. However, in our SBST integration, the LLM is only used for seeding the initial population. Since this step is performed once and not repeatedly during the search process, the longer generation time is an acceptable cost.

RQ1.

Finding 1: *Self-Refine is the most effective prompt engineering strategy to consistently produce tests with complex method sequences.*

The specific variant of Self-Refine is selected as the final strategy. While lower in median coverage than Zero-Shot-specific (63.33 vs 68.09), it outperforms other strategies in generating complex test cases and discovering edge cases, and has consistent output.

5.2. RQ2: How effective is the parser at preserving the LLM-generated test cases?

Table 5.6 shows the median branch coverage results, inter-quartile-range (IQR), and number of parsed files per class for the LLM-generated tests and their parsed counterparts. The Parsed Files column represents the number of test files (out of 10) in which the parser returned any non-empty output. It does not take into account whether the output was correct, or achieved any coverage, only whether parsing succeeded in producing a result.

On average, the parser has 10.93 percentage points lower coverage than the original tests. Over the 35 classes, 8 have a median branch coverage of 0, 13 are significantly different, with the remaining 14 having no significant difference. Interestingly, 1 of the 13 different classes was in the direction of the parsed tests, namely lists.js. This is due to the LLM consistenly calling a method that resulted in an error. This method is not part of the original class, therefore it does not appear in the list of possible targets of SynTest and does not get parsed. This 'accidental fixing' also occured in BinarySearchTreeNodee.js, min-max.js, is-moment-input and view.js, resulting in marginally higher coverage.

The parser was designed to support the most typical constructs in test cases. Currently, it can parse the following types:

- Primitive values (e.g., strings, numbers, booleans)
- Constructor calls (of the class under test)
- Method calls on constructed instances
- Function and arrow function calls
- Property access (e.g., obj.prop)
- Object creation, if all values can be mapped to supported statements

Benchmark	**File Name**	LLM		Parsed LLM		Parsed Tests	
		Median	IQR	Median	IQR	Median	IQR
Algorithms	bfTravellingSalesman.js	0.0	34.38	0.0	0.0	0.0	0.0
	depthFirstSearch.js	45.45*	61.37	0.0	40.91	0.0	65.63
	floatAsBinaryString.js	100.0	0.0	100.0	0.0	69.21	9.96
	hillCipher.js	87.50*	9.38	75.0	12.50	40.40	15.91
	Knapsack.js	75.76*	35.61	0.0	50.0	98.34	47.23
	KnapsackItem.js	50.00*	100.0	0.0	0.0	100.0	0.0
	liuHui.js	80.0	15.0	80.0	0.0	65.53	36.89
	nQueens.js	100.0	0.0	100.0	0.0	83.75	22.03
	railFenceCipher.js	100.0	0.0	100.0	0.0	79.67	12.02
	resizeImageWidth.js	0.0	0.0	0.0	4.41	81.75	32.35
	stronglyConnectedComponents.js	0.0	0.0	0.0	0.0	0.0	0.0
Data Structures	AvlTree.js	58.93	7.15	57.14	12.51	100.0	0.0
	BinarySearchTreeNode.js	97.62	4.17	98.81	4.76	93.80	79.69
	DisjointSet.js	93.75*	6.25	78.13	6.25	100.0	8.17
	DoublyLinkedList.js	75.0	10.53	73.68	5.93	100.0	0.0
	FenwickTree.js	100.00*	0.0	75.0	0.0	100.0	0.0
	GraphEdge.js	100.00*	0.0	0.0	0.0	0.0	0.0
	HashTable.js	100.0	0.0	100.0	0.0	100.0	0.0
	LinkedList.js	89.78	2.27	88.64	7.96	100.0	0.0
	Queue.js	100.0	0.0	100.0	0.0	100.0	0.0
	RedBlackTree.js	59.52	8.93	55.95	8.93	100.0	5.56
	Stack.js	100.0	0.0	100.0	0.0	100.0	0.0
	TrieNode.js	86.67	6.66	86.67	5.0	100.0	3.13
Express	layer.js	70.00*	2.50	48.34	41.66	70.72	70.13
	view.js	0.0	61.46	27.09	14.58	86.76	15.83
Moment	add-subtract.js	0.0	0.0	0.0	0.0	0.0	0.0
	bubble.js	100.00*	18.75	0.0	0.0	23.61	26.59
	compare.js	0.0	0.0	0.0	0.0	0.0	0.0
	create.js	50.00*	6.40	48.84	5.82	96.67	7.55
	from-anything.js	74.47*	5.33	70.21	3.19	82.31	11.48
	is-moment-input.js	92.50	92.50	100.0	0.0	85.79	10.99
	lists.js	0.0	0.0	77.50*	20.0	80.36	32.43
	min-max.js	0.0	0.0	13.64	13.64	10.56	20.84
	token.js	90.0	0.0	90.0	7.50	100.0	12.99
	week-calendar-utils.js	50.00*	12.50	0.0	28.13	0.0	42.19
Average		63.63	—	52.70	—	—	—

Table 5.6: Median branch coverage, Inter-Quartile Range (IQR), and number of parsed files per class over 10 runs. Statistically significant classes indicated with *

These features allow the parser to translate 12 out of 28 classes with non-zero coverage in the original LLM tests without any loss in median branch coverage. Classes where the original LLM tests already achieved 0% coverage were excluded from this count, as there was no coverage to preserve. Furthermore, the coverage manages to preserve coverage within the range $\leq 5\%$ for 19 out of 28 classes.

Examples of successful classes are HashTable.js, railFenceCipher.js, and token.js. These files consistently contain test cases with supported constructs that smoothly map to corresponding SynTest statements. An example of such a test is show in Figure 5.2a, which involves object instantiation $(\text{new HashTable}(4))$, method calls (set, get), and primitive values.

```
it('should insert and retrieve a value
    correctly', () => {
      let hashTable = new HashTable(4)
      hashTable.set('key1', 'value1');
      const result = hashTable.get('key1');
      expect(result).toBe('value1');
    });
});
```

(a) Test case with conventional patterns, leading to successful parsing

```
it('bubble transition from days to
    months', () => {
    let timeObject = {
        ...
        ...
        bubble: bubble
        };

        timeObject._days = 30;
        timeObject._milliseconds = 0;
        timeObject._months = 0;

        const result = timeObject.bubble();

        expect(result._data.months).toBe(1);
        expect(result._data.days).toBe(0);
    });
```

(b) Test case with atypical structures, leading to failed parsing

Figure 5.2: Examples of test cases where the parser succeeds and fails

Next, we investigate the classes in which the parser performs poorly and identify the underlying causes. The parser's largest drops in coverage occur in bubble.js, Knapsack.js, and layer.js.

Starting with bubble.js, this class is implemented in an unconventional way, resulting in atypical test cases. An example of which is showin in Figure 5.2b. Instead of invoking a method on a class or imported module, the tests assign the bubble function as a property to objects. They then invoke bubble on the object, causing JavaScript to dynamically bind this to that object at runtime. While this is technically valid JavaScript, it is not a typical usage patterns and breaks the parser's assumption that test methods are invoked on constructors or imported modules.

The next file is Knapsack.js, this class has an IQR of 50.00, meaning that a few of the parsed classes were actually successful in parsing. The unsuccessful runs were due to the LLM test invoking the constructor of KnapsackItem. In the list of constructor targets, SynTest only finds the constructors of the class-under-test. Therefore, the input becomes undefined and the Knapsack cannot be constructed, ultimately leading to none of the tests being parsed.

In the successful runs, the LLM created an object that has all the properties of KnapsackItem, allowing it to run all methods. While this workaround enables execution, it is technically undesirable because it introduces misleading test coverage by bypassing the actual class. The current version of the parser is unable to handle cases like this, but ideally it would recognize when auxiliary classes are invoked to be used as input. To enable this, the parser needs to query SynTest's type engine for all valid input types, rather than limiting itself to constructors from the current file.

The last class we highlight is layer.js. Within this class, there is not a singular point where the class fails, but rather multiple different deficiencies. The logs reveal various unsupported node types, such as: BinaryExpression, ThrowStatement, and FunctionExpression. These functions are uncommon, and cannot be directly mapped to existing SynTest statements. It is possible to create a workaround or implement new SynTest encodings to support these features, however the time and effort required for this is disproportional to the frequency at which they appear in test cases. Therefore, they are considered out of scope for the current prototype.

To summarize the overall parser performance, the parser preserves median branch coverage in 12 out of 27 classes with non-zero LLM-test coverage and remains within 5 percentage points in 19 out of 27 classes. In contrast, larger drops in coverage occur in cases where test code relies on features outside the parser's current capabilities, such as indirect function binding (e.g., bubble.js), use of constructors from auxiliary classes (e.g., Knapsack.js), or unsupported syntax elements (e.g., layer.js). These cases reveal specific limitations in the parser's assumptions and represent possible directions for future work to improve the parser's capabilities.

RQ2.

The parser supports primitives, constructor calls (of the class under test), method calls, function and arrow function calls, property access, and simple object creation. It preserved all coverage in 12 out of 27 classes with non-zero LLM-test coverage, and remained within 5 percentage points in 19 out of 28 classes. Failures were primarily due to unsupported features (e.g. BinaryExpression, ThrowStatements) or test code that did not align with the parser's assumptions (e.g., indirect method binding, use of auxiliary constructors).

5.3. RQ3: How effective are different test generation approaches w.r.t. branch coverage and complex structure?

Table 5.7 shows the median branch coverage every approach achieved per class over 10 runs, with the highest value per row being hightlighted in gray. Furthermore, the results of the statistical anyalsis are displayed in Table 5.8. For each pairwise comparison, the outcomes are grouped into three categories: #Lose, #No diff, and #Win, representing the number of classes where the strategy on the right performed significantly worse, showed no significant difference, or performed better, respectively. Each of these is further divided into effect size magnitudes: negligible, small, medium, and large, based on the Vargha-Delaney $\hat{A}_{12}$ metric.

Table 5.7: Median branch coverage and IQR for SynTest, LLM-tests, SynTest + LLM, LSET, and LSET + LLM configurations over 35 classes, grouped by benchmark. The LSET configuration here is set to 25 seeds, generated outside the search-time. For the LLM column the model was queried until it returned 25 tests, same as LSET

Benchmark	**File Name**	SynTest		LLM		SynTest + LLM		LSET		LSET + LLM	
		Median	IQR	Median	IQR	Median	IQR	Median	IQR	Median	IQR
Algorithms	bfTravellingSalesman.js	25.0	0.0	100.0	0.0	100.0	0.0	12.5	25.0	100.0	71.88
	depthFirstSearch.js	63.64	0.0	100.0	25.00	100.0	0.0	63.64	0.0	100.0	9.09
	floatAsBinaryString.js	100.0	0.0	100.0	0.0	100.0	0.0	100.0	0.0	100.0	0.0
	hillCipher.js	100.0	50.0	100.0	12.5	100.0	0.0	100.0	0.0	100.0	0.0
	Knapsack.js	36.36	0.0	92.43	52.28	93.94	49.25	36.36	2.27	78.79	50.76
	KnapsackItem.js	100.0	0.0	100.0	0.0	100.0	0.0	100.0	0.0	100.0	0.0
	liuHui.js	100.0	0.0	100.0	20.0	100.0	0.0	100.0	0.0	100.0	0.0
	nQueens.js	100.0	0.0	100.0	0.0	100.0	0.0	0.0	0.0	100.0	0.0
	railFenceCipher.js	100.0	0.0	100.0	0.0	100.0	0.0	100.0	0.0	100.0	0.0
	resizeImageWidth.js	0.0	0.0	0.0	0.0	0.0	0.0	2.94	5.88	2.94	58.82
	stronglyConnectedComponents.js	0.0	0.0	0.0	37.5	0.0	37.5	0.0	0.0	0.0	37.5
Data Structures	AvlTree.js	89.29	0.0	71.43	6.25	98.22	3.57	100.0	2.68	100.0	0.0
	BinarySearchTreeNode.js	59.52	4.17	98.81	2.38	100.0	2.38	94.05	2.38	100.0	0.0
	DisjointSet.js	75.0	18.75	93.75	4.69	93.75	4.69	78.13	6.25	93.75	0.0
	DoublyLinkedList.js	57.89	4.61	88.16	10.53	93.43	8.56	43.43	47.37	96.06	11.19
	FenwickTree.js	91.67	8.33	100.0	0.0	100.0	0.0	100.0	0.0	100.0	0.0
	GraphEdge.js	100.0	0.0	100.0	0.0	100.0	0.0	100.0	0.0	100.0	0.0
	HashTable.js	78.57	28.57	100.0	0.0	100.0	0.0	100.0	0.0	100.0	0.0
	LinkedList.js	76.14	5.68	96.59	2.28	100.0	0.0	100.0	0.0	100.0	0.0
	Queue.js	100.0	0.0	100.0	0.0	100.0	0.0	100.0	0.0	100.0	0.0
	RedBlackTree.js	50.0	0.0	79.76	14.88	90.48	8.34	54.76	9.52	86.91	8.94
	Stack.js	100.0	0.0	100.0	0.0	100.0	0.0	100.0	0.0	100.0	0.0
	TrieNode.js	80.0	11.67	93.33	5.0	100.0	0.0	100.0	0.0	100.0	0.0
Express	layer.js	46.67	2.5	70.0	5.83	80.0	3.33	70.0	13.34	78.34	3.33
	view.js	43.75	4.16	66.67	7.3	75.0	8.34	43.75	32.29	70.83	26.05
Moment	add-subtract.js	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	bubble.js	0.0	0.0	100.0	0.0	100.0	0.0	0.0	0.0	100.0	0.0
	compare.js	11.32	0.0	0.0	0.0	11.32	0.0	11.32	0.0	11.32	0.0
	create.js	37.21	0.0	53.49	2.32	55.81	1.74	55.81	2.33	55.81	2.33
	from-anything.js	87.23	0.0	80.85	4.26	91.49	2.13	87.23	0.0	91.49	0.0
	is-moment-input.js	100.0	0.0	100.0	0.0	100.0	0.0	90.0	8.75	100.0	0.0
	lists.js	100.0	0.0	72.5	88.75	100.0	0.0	100.0	0.0	100.0	0.0
	min-max.js	31.82	0.0	0.0	0.0	31.82	0.0	31.82	0.0	31.82	10.22
	token.js	90.0	27.5	90.0	0.0	95.0	10.0	95.0	10.0	95.0	10.0
	week-calendar-utils.js	75.0	0.0	50.0	0.0	75.0	0.0	100.0	18.75	100.0	18.75
Average		65.89		77.08		82.44		67.74		82.66	

Table 5.8: Results of the statistical analysis regarding branch coverage, comparing LSET against the baselines. Effects are categorized by size, as determined by the $\hat{A}_{12}$ metric.

Comparison	#Lose			#No diff	#Win		
	Small	Medium	Large	Negl.	Small	Medium	Large
SynTest vs. LLM	-	1	6	14	-	-	14
SynTest vs. SynTest + LLM	-	-	-	18	-	-	17
LLM vs SynTest + LLM	-	-	-	20	1	3	11
SynTest vs LSET	-	-	3	23	-	2	7
LLM vs. LSET	-	-	10	16	-	1	8
SynTest vs LSET + LLM	-	-	-	16	-	1	18
LLM vs LSET + LLM	-	-	-	21	-	2	12
SynTest + LLM vs. LSET	-	-	12	23	-	-	-
SynTest + LLM vs LSET + LLM	-	-	-	34	-	-	1

5.3.1. Baseline Comparison

We begin by comparing the results of SynTest, representing the SBST approach, with those of the LLM-based approach, using test cases generated by GPT-4o-mini. On average, LLM-tests have higher coverage than SynTest in our benchmark, with 77.08% and 65.89 %, respectively. The results of LLM-tests are generally less stable, as indicated by the high IQR in certain classes. For instance, in lists.js it reaches an IQR of 88.75%. The coverage and statistical analysis results in Table 5.8 shows that these methods excel in different classes. SynTest shows an advantage in 7 classes, while LLMs are better in 14.

This complementary characteristic is also reflected in the results of the union (SynTest + LLM) between the two approaches. When running both test suites together, the combined strategy achieves an average coverage of 82.436%. Coverage is significantly higher in 17 classes compared to SynTest alone, and in 16 classes compared to the LLM-tests.

SynTest's main advantage lies in its systematic exploration of edge cases and diverse inputs. It generates a wide range of unexpected inputs, such as null, incorrect types, and invalid input arrays, that help uncover branches related to error handling that LLMs commonly leave uncovered. In AvlTree.js, for example, SynTest successfully triggered rotation branches involving non-root nodes with missing or incomplete children, which were missed by LLM-generated tests that assumed structurally valid trees. Similarly, in classes like Moment's minMax.js, SynTest covered branches by passing incomplete or incorrect values, such as arrays of non-moment objects or empty inputs, leading to internal errors.

The cases where LLM outperforms SBST can be attributed to two reasons: better test structure and complex object construction. In classes like AvlTree.js, DisjointSet.js, and DoublyLinkedList.js it manages to chain many method invocations, such as combining multiple insert and remove operations in AvlTree, that can trigger deeper branches.

Additionally, LLMs are capable of constructing complex objects that required to call some methods. For example, in bubble.js, it generated the following setup: For example,in bubble.js it created the following object:

```
instance = {
            _milliseconds: 0,
            _days: 0,
            _months: 0,
            _data: {
                milliseconds: 0,
                seconds: 0,
                minutes: 0,
                hours: 0,
                days: 0,
                months: 0,
                years: 0,
            },
            bubble: bubble,
            daysToMonths: daysToMonths,
            monthsToDays: monthsToDays};
```

Such an object is very difficult for SBST to construct, as the lack of type information makes the size of the search space too large.

Additionally, LLMs were able to produce realistic strings that are required for certain branches, like in Layer.js

```
it('should correctly extract multiple path parameters', () => {
  const multiParamLayer = new Layer('/user/:id/profile/:profileId', {}, mockFn);
  const result = multiParamLayer.match('/user/123/profile/456');
  expect(result).toBe(true);
  expect(multiParamLayer.params).toEqual({ id: '123', profileId: '456' });
  expect(multiParamLayer.path).toBe('/user/123/profile/456');
});
```

Because SBST will randomly produce strings, it can not hit the true branch of match, while LLMs natural language capabilities allow it to do so.

RQ3.1

Finding 2: *SBST and LLM-based approaches are complementary since they cover different sets of branches.*

SynTest achieved 65.89% branch coverage on average, while LLM managed to cover more with 77.08% coverage. SBST and LLM-generated tests have complementary strengths regarding branch coverage. SBST excels in systematically exploring edge cases, particularly involving invalid inputs, which uncover error-handling branches. In contrast, LLM-generated tests are superior at creating method call sequences and constructing complex objects or realistic strings. This is also reflected in their union,with SynTest + LLM having higher average coverage (82.44%) than each approach separately.

5.3.2. LSET Compared to the Baselines

The LSET approach achieves an average coverage of 67.74%, placing it between SynTest and LLM-tests. However, when the original LLM-generated tests are included alongside the translated tests used for seeding, the combined approach (LSET + LLM) reaches the highest average coverage of 82.66%. The gap between these two results highlights the limitations of the current parser, which fails to translate certain valid LLM-generated tests into executable SBST seeds, resulting in coverage loss.

The reported results for LSET specifically reflect the configuration using 25 initial seeds, with LLM prompting conducted outside the search time. This setting provides the best coverage when considering results without adding the original LLM tests. Upon integrating the original LLM tests, using 50 initial seeds slightly outperforms or matches the 25-seed configuration in most cases, but due to the minimal incremental coverage gain and significantly higher computational cost, the 25-seed configuration remains the practical choice for comparison.

An abnormal coverage result is observed with the nQueens.js class, as it has zero coverage despite the baselines both having 100%. Closer inspection reveals that this class consistently triggered implementation errors. Apart from this case, LSET generally showed no consistent implementation errors.

Table 5.9: Comparison of SynTest, LLM, and LSET for Method Structure Metrics

Strategy	Average Consecutive Calls			Max Consecutive Calls		
	Mean	**Median**	**Std**	**Mean**	**Median**	**Std**
SynTest	1.33	1.33	0.04	2.27	2.26	0.09
LLM	2.13	2.15	0.09	4.55	4.41	0.29
LSET	1.44	1.38	0.11	2.82	2.76	0.28

Alongside branch coverage, we also measured consecutive method calls for each approach as shown in Table 5.9. This metric serves as an indication for the complexity of each approach's method sequences.

LSET performs better than SynTest in both average and maximum consecutive calls. The average for LSET is 1.44 compared to 1.33 for SynTest, while the maximum is 2.82 versus 2.27. This suggests that seeding the search with LLM-generated tests results in the final test suite containing tests with more complicated structure. Still, LSET doesn't reach the same level as the standalone LLM tests, which achieve a much higher average (2.13) and maximum (4.55).

The average of LSET being more similar to SynTest is expected as the final test suite also contains entries that evolved from unseeded tests. However, the max being substantially lower is due to two reasons. First, the original structure of LLM-generated tests is sometimes not fully translated due to limitations in the parser. Second, even when translation succeeds, the original LLM-structure is not guaranteed to survive the evolutionary process.

This structural variation in Figure 5.3 which displays three tests that LSET generated for layer.js. For example, the first test is an exact copy of the LLM test, causing it to have clear structure. In contrast, the next test, contains minimal structural logic and appears to have been evolved purely for coverage. In between these extreme cases, there are also hybrids which use the structure provided by the seeds to cover more branches. In this example, the original seeds did not find the error with unmatched)

This structural variation is illustrated in Figure 5.3, which displays three test cases generated by LSET for layer.js. The first test is a one-to-one copy of an LLM-generated seed, preserving its clear and meaningful structure. It demonstrates a typical usage scenario with descriptive variable names and a logical sequence of calls. In contrast, the second test represents a pure SBST-style case, shown by arbitrary input values and structure in order to trigger an exception. In between these extreme cases, the third test showcases a hybrid example, where the LLM-provided structure is utilized to reach new branches. In this case, the original LLM seeds did not discover the unmatched parenthesis in the path string, but the hybrid test successfully triggers this error with a clear structure.

One-to-one LLM-Structured Test

```
const path = "/test/:id";
const options = {}
const fn = (req, res, next) => {};
const layer = new Layer(path, options, fn)
const path1 = "/test/123";
const matchReturnValue = await layer.match(path1)

// Assertions
expect(matchReturnValue).to.equal(true)
  })
```

SBST Style Test

```
const arrayElement = null;
const arrayElement1 = 204;
const arrayElement2 = {}
const path = [arrayElement, arrayElement1,
arrayElement2]
const options = true;
const localName = false;
const fn = {
    "name": localName
}

// Assertions
await expect((async () => {
  new Layer(path, options, fn)
})()).to.be.rejectedWith("Cannot read properties of null (reading 'length')")
  })
```

Hybrid Test (LLM + SBST)

```
const path = "/test/)id";
const options = {}
const fn = (req, res, next) => {};

// Assertions
await expect((async () => {
    new Layer(path, options, fn)
})()).to.be.rejectedWith("Invalid regular expression: /^\\/test\\/)id\\/?$/i: Unmatched ')'")
```

Figure 5.3: Examples of structural variation in LSET tests

Table 5.8 summarizes the pairwise statistical comparisons regarding branch coverage between LSET and the baselines. Comparing LSET to SynTest alone, LSET achieves significantly better coverage in nine classes, significantly worse in three classes and no difference in 23. When compared to the standalone LLM-generated tests, LSET performs significantly better in nine classes although it also significantly loses in ten classes. Finally, when analyzing LSET against the combined SynTest and LLM suite (SynTest + LLM), LSET loses in 12 classes and does not manage to perform better in any class.

However, when including the original LLM tests used for generating seeds (LSET + LLM), the outcome differs notably, with LSET + LLM consistently achieving equal or superior coverage compared to all other methods. Nevertheless, the difference between SynTest + LLM and LSET + LLM is minimal, with LSET + LLM only outperforming the former in a single class.

SynTest outperformed LSET in three classes: bfTravellingSalesman.js, nQueens.js, and is-moment-input.js. In nQueens.js, the difference is caused by an implementation error, as discussed previously. In the case of bfTravellingSalesman.js, there are causes. First, due to runtime errors during test generation, only 5 usable test suites were produced across 10 runs. Second, even in these successful runs, the parser was unable to translate any of the LLM-generated tests, as they relied on constructors from external classes which the current parser does not support. As a result, no effective seeding occurred, meainng that LSET is effectively the same as the random sampler.

For is-moment-input.js, the lower coverage stems from a difference in coverage instrumentation. SynTest uses its own AST visitor and Mocha for measuring coverage, whereas we report coverage from Jest, which instruments the code differently and identifies a different set of branches. The logs show that LSET evaluates exactly 50 tests, the size of the initial population, and reports 4 out of 4 branches covered, meaning the algorithm terminated immediately after initializing the population. The fact that no evolution took place is also shown the final test suites as most tests are directly parsed from the LLM without any modifications. Therefore, the performance gap is not a flaw in the approach but a consequence of measurement differences.

Despite these exceptions, LSET was able to introduce improvements in several classes. For instance, better test structure was achieved in FenwickTree.js, HashTable.js, and TrieNode.js. Additionally, more realistic input strings were produced in classes like create.js, hillCipher.js, and layer.js, leading to significantly better coverage compared to SynTest.

The single class where LSET + LLM was significantly better than SynTest + LLM, week-calendar-utils.js, showcases the potential of the LSET approach. In particular, LSET was the only method consistenly able to trigger two branches in the weekOfYear function that remained untested by both SynTest and standalone LLM approaches.

The weekOfYear function takes as input a date-like object mom, along with two integers: dow (day of week) and doy (day of year that starts the first week). Internally, it computes which week of the year a given day falls into. The function contains two important branches:

1. **Underflow branch**:

```
if (week < 1) {
  resYear = mom.year() - 1;
  resWeek = week + weeksInYear(resYear, dow, doy);
}
```

   This branch is taken when the computed week is less than 1. This can happen if the given dayOfYear falls into the previous year's final week, typically because doy is large or dayOfYear() is very small.

2. **Overflow branch**:

```
else if (week > weeksInYear(mom.year(), dow, doy)) {
  resWeek = week - weeksInYear(mom.year(), dow, doy);
  resYear = mom.year() + 1;
}
```

   This branch occurs when the computed week exceeds the total number of weeks in the year. This can happen when dayOfYear() is unusually large or the doy is small, effectively pushing the calculated week into the next calendar year.

SynTest failed to trigger both conditions due to invalid or overly random inputs. For example, some test cases passed a number as mom, which caused runtime exceptions:

```
// SynTest example failing due to invalid mom object
const mom = 760.08414411151883;
await weekOfYear(mom, undefined, 50); // Throws "mom.year is not a function"
```

Meanwhile, LLM-generated tests used valid mocks but still did not produce the right input combinations to cross either boundary. For example:

```
// LLM example that doesn't exceed weeksInYear boundary
const mockDate = { year: () => 2023, dayOfYear: () => 366 };
const result = weekOfYear(mockDate, 1, 4);
expect(result).toEqual({ year: 2024, week: 1 });
```

Although this test uses the maximum possible dayOfYear value, the combination of dow and doy did not cause the computed week to exceed the year boundary.

In contrast, LSET seeded the search with valid LLM mocks and then used evolutionary search to evolve values like dayOfYear, dow, and doy. Eventually, it discovered configurations that pushed the computed week into both underflow and overflow conditions:

```
// LSET example triggering overflow branch
const mockDate = { year: () => 2023, dayOfYear: () => 366 };
const overflowResult = await weekOfYear(mockDate, 1, 1);
expect(overflowResult).to.deep.equal({ week: 1, year: 2024 });
```

By combining the LLM's superior test structure with SBST's systematic exploration, LSET was consistently able to cover all branches—demonstrating its potential to be greater than the sum of its parts.

RQ3.2

Finding 3: *Seeding using LLM-generated tests can improve SBST test suites, both in coverage and test structure*

LSET achieved an average coverage of 67.74%, which is higher than SynTest but lower than the LLM-generated tests. When combined with the original LLM tests used for generating seeds (LSET + LLM), the approach attained the highest average coverage of 82.66%. LSET also outperformed SynTest in average and maximum consecutive method calls but remained below the LLM in this metric. In direct comparisons, LSET achieved significantly better coverage than SynTest in 9 classes, and significantly worse in 3. The wins are primarily due to better test structure and more realistic input data. Compared to the LLM, it was better in 9 classes, but worse in 10. Anst the combined SynTest + LLM suite, LSET was worse in 12 classes and did not outperform it in any. The LSET + LLM combination outperformed SynTest + LLM in one class and was equal in all others.

5.3.3. Similarity of LLM Tests to Manual Tests

Since our benchmark consists of popular open-source JavaScript projects, there is a very high likelihood they are present in the LLMs training set, creating the risk of **data leakage**. Therefore, there is a risk of potentially inflated performance due to pretraining on the test set [33]. To investigate whether the LLM is reusing memorized patterns or is capable of generating novel tests, we conduct a similarity analysis between LLM-generated and manually written tests.

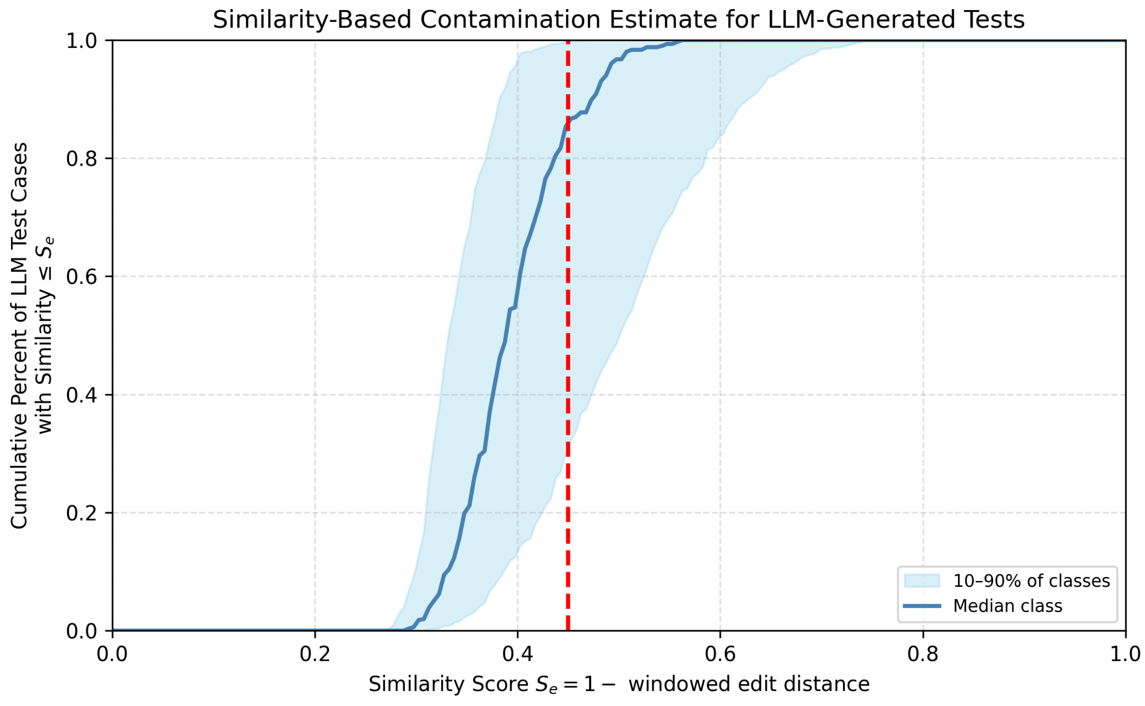


Figure 5.4: Overall similarity distribution across classes. The line shows the median class. The shaded area shows the 10th–90th percentile range across all classes.

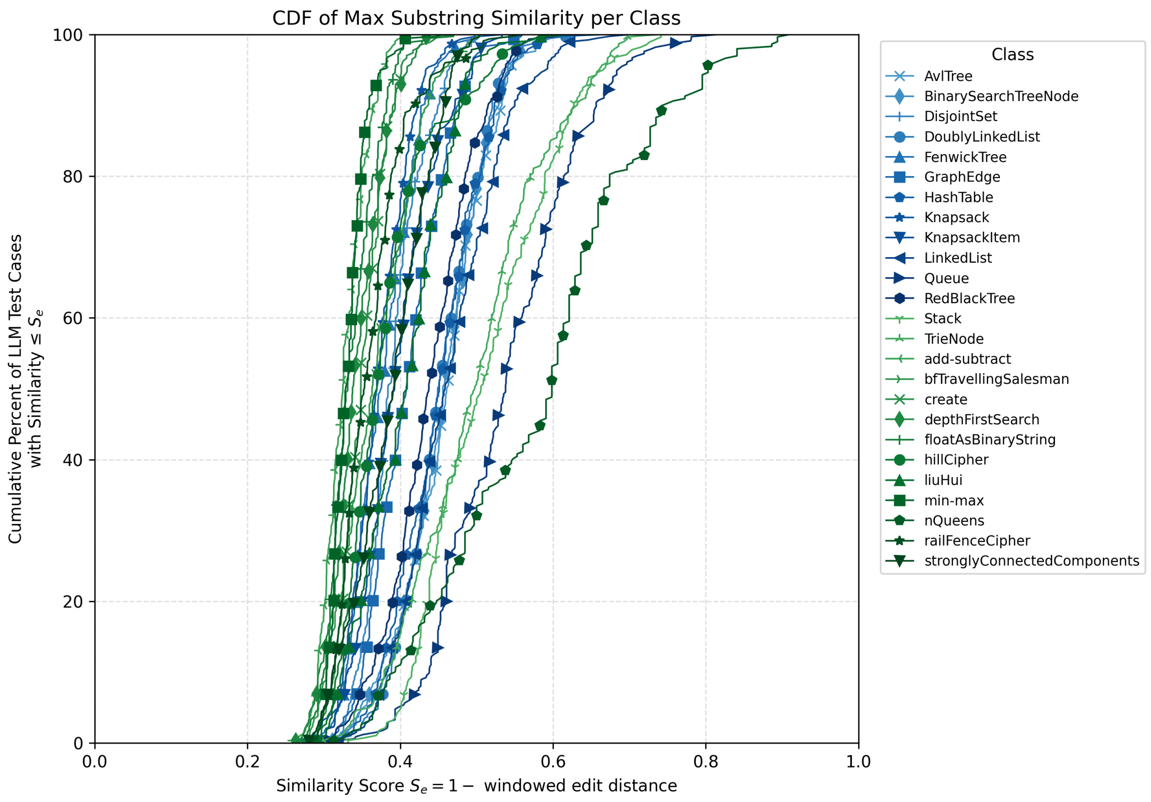


Figure 5.5: Similarity scores for each test class. Each line shows the cumulative distribution of maximum substring similarity between LLM-generated and manual tests in a given class.

Figure 5.4 shows a cumulative distribution envelope of maximum substring similarity scores across all classes. For each similarity threshold S_e, the graph plots the cumulative fraction of LLM-generated tests with similarity less than or equal to S_e, aggregated across classes. The solid line represents them median class, while the shaded region spans the 10th to 90th percentile of per-class CDFs.

Furthermore, Figure 5.5 displays the cumulative distribution of similarity scores for each individual class with a reference manual test suite. Each line corresponds to a distinct class and shows how similarity patterns vary across cases. Most classes follow a steep rise between $S_e = 0.3$ and $S_e = 0.5$, while a few outliers (e.g., Queue, $\mathrm{nQueens}$) extend well beyond $S_e = 0.6$, indicating possible memorization.

To gain a better understanding of what a similarity score represents in practice, we now analyze representative LLM-generated and manual test cases across low, high, and borderline similarity levels.

```
let momentInstance;

beforeEach(() => {
    // Create a new moment instance for
         testing
    momentInstance = moment('2023-01-01');
});

describe('Adding durations', () => {
    it('should add days correctly', () => {
        add.call(momentInstance, 5, 'days');
        expect(momentInstance
        .format('YYYY-MM-DD'))
        .toBe('2023-01-06');
    });

    it('should add months correctly', () => {
        add.call(momentInstance, 2, 'months');
        expect(momentInstance
        .format('YYYY-MM-DD'))
        .toBe('2023-03-01');
    });

    it('should add multiple types of
         durations correctly', () => {
        add.call(momentInstance, 1,
             'months'); // from January to
             February
        add.call(momentInstance, 10, 'days');
              // February 1 to February 11
        expect(momentInstance.
        format('YYYY-MM-DD')).
        toBe('2023-02-11');
    });
```

(a) LLM-generated tests for add-subtract, all with low similarity (<0.32)

```
test('add long reverse args', function
     (assert) {
    var a = moment();
    a.year(2011);
    a.month(9);
    a.date(12);
    a.hours(6);
    a.minutes(7);
    a.seconds(8);
    a.milliseconds(500);

    assert.equal(
        a.add({ milliseconds: 50
             }).milliseconds(),
        550,
        'Add milliseconds'
    );
    assert.equal(a.add({ seconds: 1
         }).seconds(), 9, 'Add seconds');
    assert.equal(a.add({ minutes: 1
         }).minutes(), 8, 'Add minutes');
    assert.equal(a.add({ hours: 1 }).hours(),
         7, 'Add hours');
    assert.equal(a.add({ days: 1 }).date(),
         13, 'Add date');
    assert.equal(a.add({ weeks: 1 }).date(),
         20, 'Add week');
    assert.equal(a.add({ months: 1
         }).month(), 10, 'Add month');
    assert.equal(a.add({ years: 1 }).year(),
         2012, 'Add year');
    assert.equal(a.add({ quarters: 1
         }).month(), 1, 'Add quarter');
});
```

(b) Example test case found in the manual test suite

Figure 5.6: Comparison of LLM-generated and manual tests for add-subtract, a low-similarity case

The add-subtract class consistently exhibits low similarity scores between LLM-generated and manual tests. As shown in Figure 5.6, the LLM-produced tests differ notably in structure and scope There is minimal overlap in assertions, order, or size. This suggests that in this case, the LLM did not reproduce the corresponding manual test suite.

```
// High similarity case (0.78)
test('multiple enqueue and dequeue operations should return elements in correct order', () => {
    queue.enqueue(1);
    queue.enqueue(2);
    queue.enqueue(3);

    expect(queue.dequeue()).toBe(1);
    expect(queue.dequeue()).toBe(2);
    expect(queue.dequeue()).toBe(3);
    expect(queue.isEmpty()).toBe(true);
});

// Low similarity case (0.39)
test('should handle a large number of enqueues and dequeues', () => {
    const numbers = Array.from({length: 1000}, (_, i) => i + 1);
    numbers.forEach(num => queue.enqueue(num));
    for (let i = 0; i < 500; i++) {
        queue.dequeue(); // Dequeue half of them
    }
    expect(queue.peek()).toBe(501); // Check head after some dequeues
    expect(queue.size()).toBe(500); // Assume there's a size method to check size
  });
```

(a) Two LLM-cases. First test has high similarity (0.78), while the second indicates novel behavior (0.39 similarity)

```
it('should dequeue from queue in FIFO order', () => {
    const queue = new Queue();

    queue.enqueue(1);
    queue.enqueue(2);

    expect(queue.dequeue()).toBe(1);
    expect(queue.dequeue()).toBe(2);
    expect(queue.dequeue()).toBeNull();
    expect(queue.isEmpty()).toBe(true);
});
```

(b) Corresponding manual test case for the high-similarity test

Figure 5.7: Comparison of LLM-generated and manual tests for Queue.js

The queue class displays a wider range of similarity scores, with both close matches and more distinct cases. Figure 5.7 contrasts a high-similarity LLM test that closely matches the structure and content of a manual FIFO test, and another that differs in scale and purpose. While we cannot rule out exposure to similar examples during training, the divergence in some cases indicates that the model can generate tests that are not trivially linked to the reference suite.

To distinguish between structurally distinct and potentially memorized test cases, we define a similarity threshold of 0.45. This value captures approximately 70.7% of all LLM-generated tests while excluding all examples from the add-subtract and min-max classes, which exhibit the lowest overall similarity and appear clearly distinct from the manual suites. In contrast, raising the threshold to 0.5 would include over 84% of tests, risking the inclusion of more overlapping cases, while lowering it to 0.4 would exclude nearly half the dataset, and potentially risk many false positives. Furthermore, the choice of 0.45 corresponds to a soft inflection point in the cumulative similarity distribution (see Figure 5.4). Note that the graph reflects per-class cumulative distributions, where each class contributes equally. This explains the visual estimate of 85% at the threshold, which is higher than the overall 70.7% obtained by weighting all test cases equally.

```
test('should create disjoint sets for unique
    values', () => {
    disjointSet.makeSet(1);
    disjointSet.makeSet(2);
    expect(disjointSet.find(1)).toBeTruthy();
    expect(disjointSet.find(2)).toBeTruthy();
});
```

(a) Boundary LLM-case for DisjointSet.js with 0.46 similarity

```
it('should do basic manipulations on disjoint
    set', () => {
    const disjointSet = new DisjointSet();

    expect(disjointSet.find('A')).toBeNull();
    expect(disjointSet.find('B')).toBeNull();

    disjointSet.makeSet('A');

    expect(disjointSet.find('A')).toBe('A');
    expect(disjointSet.find('B')).toBeNull();

    disjointSet.makeSet('B');

    expect(disjointSet.find('A')).toBe('A');
    expect(disjointSet.find('B')).toBe('B');
```

(b) Corresponding excerpt from manual test case

Figure 5.8: Example of a borderline case ($S_e = 0.46$), where the LLM-generated test is structurally similar to part of the manual suite but differs in syntax and level of detail.

To evaluate the effectiveness of this threshold, we examine a borderline case with a similarity score of 0.46, just above the cutoff. As shown in Figure 5.8, the LLM-generated test and its manual counterpart share some structural traits, but differ in size and assertions. This illustrates how cases around this value can contain structural similarities without being a direct copy.

RQ3.3.

LLMs can generate tests that differ meaningfully from reference suites, though overlaps exist. A threshold of 0.45 flags roughly 70% of LLM-generated tests as non-similar to tests cases in manually-written reference suites.

5.4. RQ4: How do different configurations of the LSET approach impact test generation performance w.r.t. branch coverage

In this section we evaluate the effects of different LSET configurations, namely number of seeds and prompting the LLM within- and outside the search-budget.

5.4.1. Effect of Number of Seeds

Table 5.10: Results of statistical analysis w.r.t. branch coverage. Effect of seed size. Top half is outside search time, bottom half is inside

Comparison	#Lose			#No diff	#Win		
	Small	Medium	Large	Negl.	Small	Medium	Large
LSET-10 vs LSET-25	-	-	-	31	-	-	4
LSET-10 vs LSET-50	-	-	2	30	-	-	3
LSET-25 vs LSET-50	-	-	2	33	-	-	-
LSET-10 + LLM vs LSET-25 + LLM	-	-	-	31	-	-	4
LSET-10 + LLM vs LSET-50 + LLM	-	-	-	29	-	-	6
LSET-25 + LLM VS LSET-50 + LLM	-	-	-	34	-	-	1
LSET-10 vs LSET-25	-	-	-	34	-	-	1
LSET-10 vs LSET-50	-	-	2	32	-	-	1
LSET-25 vs LSET-50	-	-	4	30	-	-	1
LSET-10 + LLM vs LSET-25 + LLM	-	-	-	32	-	1	2
LSET-10 + LLM vs LSET-50 + LLM	-	-	-	29	-	-	6
LSET-25 + LLM VS LSET-50 + LLM	-	-	-	34	-	-	1

Table 5.10 shows the result of the statistical analysis for all configurations. When not taking account the LLM tests, 25 seeds is the best configuration, both when querying the LLM in- and outside search-time.

However, LSET-50 is superior in 1 class and equal in all other when taking account the LLM cases.

RQ4.1

Finding 4: *A 50/50 split between seeded tests and random tests in the initial population offers the best balance between effectiveness and cost*

LSET-25 has the best performance, regardless of whether the LLM is queried inside or outside of search-time. When including the LLM-tests, 50 seeds had marginally better performance over LSET-25 (better in 1 class, equal in 34), with the trade-off of higher computation costs.

5.4.2. Effect of Querying Inside- vs Outside Search Time

Table 5.11 shows the head-to-head comparison for each seed configuration when prompted in- and outside search-time. Surprisingly, there are four cases where the inside version wins. Three out of four cases where in layer.js, with the fourth case being bfTravellingSalesman.js. This is an unexpected result as all variables are the same, except that the outside versions strictly have more time.

Overall, prompting the LLM outside of search-time yields slightly better performance on average, but the difference between the two configurations is relatively minor (max. 2 classes). This indicates that allocating more search-time does not necessarily lead to higher coverage in LSET, suggesting that coverage tends to plateau early in the search process. It also demonstrates that prompting inside the search-time can still be viable, if runtime resources want to be prioritized.

The average time spent querying the LLM for 10, 25, and 50 seed configurations was 52.31s, 68.36s, and 79.44s respectively, as shown in Table 5.11. These values suggest that the batch size used for LLM queries was set too conservatively. Ideally, the time should remain roughly constant across configurations, since queries are executed in parallel. However, for higher seed sizes, the time increased due to not generating enough tests in the initial batch, which triggered additional rounds of querying to meet the desired number of seeds.

Table 5.11: Results of statistical analysis w.r.t. branch coverage. Effect of search-time

Comparison	#Lose			#No diff	#Win		
	Small	Medium	Large	Negl.	Small	Medium	Large
LSET-10-IN vs LSET-10-OUT	-	-	1	34	-	-	-
LSET-10-IN + LLM vs LSET-10-OUT + LLM	-	-	1	33	-	1	1
LSET-25-IN vs LSET-25-OUT	-	-	1	33	-	-	1
LSET-25-IN + LLM vs LSET-25-OUT + LLM	-	-	-	34	-	-	1
LSET-50-IN vs LSET-50-OUT	-	-	-	33	-	1	1
LSET-50-IN + LLM vs LSET-50-OUT + LLM	-	-	1	34	-	-	-

Table 5.12: Summary of LLM query-time for each LSET seed size configuration

Seed Size	**Mean**	**Min**	**Max**
LSET-10	52.31	33.72	89.78
LSET-25	68.37	32.99	157.20
LSET-50	79.45	43.34	195.49

RQ4.2.

Finding 5: Querying LLMs inside or outside the search-time yields similar results

Prompting the LLM outside of search-time yields marginally better results, but the difference compared to inside-search prompting is small. This suggests that allocating more search-time does not necessarily result in higher coverage. Therefore, prompting within search-time can be a viable alternative when runtime resources are limited.

6 Discussion

6.1. Interpretation of LSET Results

This section analyzes the empirical results of the LSET approach in terms of test structure and coverage. While LSET aims to combine the structure of LLM-generated tests with the systematic exploration of SBST, the results show mixed success. We observe both structural variation within the test suites and modest coverage improvements. The following subsections examine these findings in detail.

6.1.1. Structural Preservation During Test Evolution

While the LSET approach clearly benefits from LLM-seeded tests, the extent to which their structure is preserved varies greatly across the test suite. As shown in the results (Section 5.3.2) and illustrated in Figure 5.3, test cases within the final suite can be categorized into three structural types: one-to-one preserved LLM structures, purely SBST-evolved tests, and hybrid cases.

```
const path = "/test/)id";
const options = {}
const fn = (req, res, next) => {};

// Assertions
await expect((async () => {
  new Layer(path, options, fn)
})()).to.be.rejectedWith("Invalid regular
    expression: /^\\/test\\/)id\\/?$/i:
    Unmatched ')'")
```

(a) LSET Test

```
const path =
    "7_s6)N\"/Kg\n*F`kq;%N5\\U@i+I3B{E2nBE{";
const end = false;
const options = {
  "end": end
}
const arrayElement = "hqq5`VE\t^-zyLm@Pv)...";
const arrayElement1 = null;
const fn = [arrayElement, arrayElement1]

// Assertions
await expect((async () => {
  new Layer(path, options, fn)
})()).to.be.rejectedWith("Invalid regular
    expression: /^7_s6)N...Unmatched ')'")
```

(b) SynTest Test

Figure 6.1: Comparison between an LSET hybrid test (a) and a pure SBST test (b) that both target the same unmatched parenthesis error in layer.js. The LSET test preserves semantic structure, while the SBST test relies on arbitrary inputs.

The most interesting group consists of the hybrid cases. These hybrid tests retain the structure provided by the LLM seed and are further evolved to reach new objectives. For instance, Figure 6.1 shows a hybrid test that successfully triggers a regular expression parsing error while maintaining clean structure. The added value of LSET becomes clear when comparing to a traditional SBST test that covers the same objective. The LSET test is interpretable and realistic, whereas the SBST version is compilable valid but unreadable. This example showcases that LSET's benefits are not limited to an increase in coverage, a test's structure can be meaningfully improved even when covering the same objective.

However, not all hybrid tests are equally successful. In some cases, the evolutionary process introduces irrelevant code fragments, confusing variable interactions, or structural inconsistencies. These tests no longer resemble the original seed in any meaningful way and offer little advantage over traditional SBST tests. As a result, while hybridization enables the possibility of producing high-quality tests, it does not guarantee it.

This observation suggests an important direction for future work: identifying mechanisms to increase the likelihood of generating useful hybrids. We discuss possible solutions in Section 6.3, including the idea of explicitly optimizing for structural quality and designing new operators that account for interdependent methods.

6.1.2. Explaining LSET's Limited Coverage Gains

While the LSET approach was designed to leverage the strengths of both LLM-generated seed tests and evolutionary search, its improvements over the simpler SynTest + LLM combination were limited. In fact, LSET + LLM only achieved significantly higher coverage in one out of 35 benchmark classes. The primary reason for LSET's limited overall effectiveness lies in the difficulty distribution of the benchmark: some classes were too easy, while others were too hard.

- In 19 out of 35 cases, no improvement was possible, as all branches were already covered.
- In at least 5 classes, the LLM-generated test suites were structurally flawed or targeted incorrect behavior (e.g., compare.js, min-max.js), making the evolutionary process ineffective at building upon them.
- In cases where the parser was unable to correctly translate certain code constructs, evolution could not take place. This effectively nullifies the benefits of seeding.

When analyzing uncovered branches, the overlap between LSET + LLM and SynTest + LLM was remarkably high. Of the 1,417 branches missed by LSET + LLM and the 1,386 missed by SynTest + LLM, 1,296 were shared. This indicates that both approaches consistently struggle with the same parts of the code and that LSET does not significantly expand the search space beyond what SynTest already explores.

For example, the following branch in create.js is missed by all approaches:

```
if (isDuration(input) && hasOwnProp(input, '_locale')) {
    ret._locale = input._locale;
}
```

This branch is only executed when createDuration is called with a Duration object that has an own _locale property. Such inputs are not generated by the LLM, and the search space is too large for the evolutionary algorithm to randomly mutate them into existence. This illustrates a key limitation of the LSET strategy: if the LLM fails to cover a branch, the evolutionary process alone often cannot compensate.

Finding 6: *LSET is only as effective as the quality of its seeds, If they lack the necessary structure to reach key branches, the evolutionary process cannot compensate.*

To better understand the scenarios where LSET does provide meaningful benefits, we conducted a targeted follow-up experiment on a carefully selected subset of classes. This mini-experiment was designed to isolate cases where both LLM and search-based methods fail, allowing us to observe whether LSET can successfully compensate for these limitations.

6.2. Isolating the Added Value of LSET

While LSET did not show substantial gains across the full benchmark set, this does not imply that the approach lacks value. Instead, its effectiveness may be limited by the benchmark itself. To better understand when LSET contributes meaningfully to test coverage, we conducted a targeted follow-up experiment designed to isolate scenarios where its evolutionary capabilities are most likely to help.

The core idea behind this experiment was to examine whether LSET can compensate for coverage gaps in cases where both LLM-generated tests and search-based methods fail. However, in many classes the LLM alone performs too well, making it difficult to observe any added benefit from evolution. To address this, we artificially reduce the LLM's coverage by taking a random subset of 5 from LLM-generated test suites. Thereby creating an opportunity for LSET to compensate for the lost coverage.

We selected two classes for this experiment: LinkedList.js and layer.js. These were chosen because SynTest consistently struggled to reach certain branches in both, and the generated LLM test code could be parsed without errors, allowing LSET to run effectively.

The results are summarized as follows. For layer.js, the LLM subset achieved 20% coverage, while full LLM coverage was previously measured at 70%. SynTest alone reached 46.67%, and LSET, seeded with the reduced LLM subset, achieved 43%. For LinkedList.js, the LLM subset covered 50%, compared to 96.69% for the full LLM and 76.135% for SynTest. LSET improved the subset to 84.09%. Note that LLM and SynTest results were taken from previous full experiments and averaged over 10 runs, whereas LSET was run 3 times in this focused setup.

Looking more closely, LSET was able to reach specific branches missed by both the subset and SynTest. For example, in LinkedList it covered this branch in the deleteTail method:

```
// deleteTail method
if (this.head === this.tail) {
  // There is only one node in linked list.
  this.head = null;
  this.tail = null;
}
```

This branch handles the case where the linked list contains exactly one node. Covering it requires generating a list with a single element and then triggering a tail deletion. In our experiment, SynTest missed this branch 7 out of 10 times, while LSET covered it in all 3 runs. This suggests that LSET's evolutionary process can discover difficult branches, but only when the initial seeds provide a sufficient foundation

In layer, similar observations were made. For instance:

```
// fast path for * (everything matched in a param)
if (this.regexp.fast_star) {
  this.params = { '0': decode_param(path) };
  this.path   = path;
  return true;
}
```

Listing 6.1: Fast-star shortcut in Layer.prototype.match

This branch is only taken when the route pattern is exactly $*$ and the path-to-regexp library sets the fast_star flag. Generating this combination requires constructing a router layer with the literal pattern $*$ and invoking match(path) with any request path. Neither the LLM subset nor SynTest created this setup, yet LSET reached it in all 3 focused runs. This illustrates how LSET can exploit even minimally structured seeds to evolve inputs toward narrow, edge-case behavior.

These examples showcase the core idea of LSET: evolving test cases containing meaningful structure can lead to increased coverage compared to traditional search. However, total coverage remained lower than the full LLM-generated test suites, reiterating LSET's weakness in reconstructing any structure not already present in the seeds. Nevertheless, this experiment demonstrates that LSET can provide added value in closing coverage gaps.

6.3. Future Work

While our results demonstrate the potential of the LSET approach, multiple future work directions remain to enhance its components and maximize its effectiveness.

Extracting and Leveraging Linkage-Structures

One primary limitation identified in our experiments is the structural degradation of complex method sequences during the evolutionary process. This may be addressed by the alternative approach discussed in Section 3.2.2 which proposed explicitly learning and preserving interdependent methods as linked structures. By incorporating these linkages into the evolutionary process, we can ensure that the LLM-derived structures are actively preserved and maintained over the process.

Compared to seeding, this approach has the advantage that the desired structure is actively maintained and evolved, rather than being introduced at initialization and potentially discarded. However, a major challenge is the risk of overfitting, where repeated use of a limited set of linked sequences may reduce population diversity and hinder the discovery of novel behaviors. Future work must investigate how to mitigate this risk and strike a balance between structural bias and exploratory diversity.

Optimizing for Structure

Another potential way to retain test structure is to incorporate it as a secondary objective in the search process. Currently, in SynTest, when two tests cover the same objective, the shorter one is preferred. While this can work in practice, it can also penalize tests that are slightly longer but structurally superior. Daka et al. addressed a similar issue by introducing readability as a secondary objective in EvoSuite, using a machine learning model trained on human-labeled data [10]. In our context, a structure-aware metric could be defined to reward coherent, logically ordered method calls. Incorporating such a measure into the multi-objective search would allow the framework to prefer not just shorter tests, but ones that are also more understandable and maintainable.

Test Case Readability

One strength of LLMs is their ability to produce readable and natural-looking code. LSET inherits some of this structure, through the use of seeding. Future work could explore the impact of LSET on test readability understandability, via human studies.

Prompt Engineering

While our evaluation of eight prompt variants provides an initial foundation, prompt engineering remains a wide and active research area. Future work could explore more advanced strategies such as evolutionary prompting [31], where prompts are optimized based on feedback from test performance. Another possible direction is dynamic prompting, where partial coverage data from earlier tests is embedded into follow-up prompts to steer the LLM toward unexplored branches.

Parser Improvements

The current parser limits LSET's effectiveness by failing to support many JavaScript specific constructs, especially higher-order functions and external class dependencies. Additionally, one potential enhancement is to fall back to a dynamic call. Instead of skipping untranslatable code, we can copy the statement in its entirety into the test by encoding it as this dynamic call statement. Although this statement could not be mutated, as SynTest does not recognize its type, it still contributes to coverage. This fallback mechanism would allow the parser to function for almost all constructs, even without exhaustively supporting all node types.

Alternate Language Models

The effectiveness of LSET may vary significantly with different LLMs. In this thesis, we used GPT-4o-mini, which is relatively small. Larger models (e.g., GPT-4o-turbo, Claude 3.7) or open-source alternatives could result in different outcomes.

Broader Benchmark
Our evaluation focused on a small benchmark with a focus on high-cohesion classes. To better understand generalizability, future studies should apply LSET to larger benchmarks. Especially, datasets that are not in the LLM model's training set are valuable to mitigate data leakage concerns.

6.4. Limitations

The results presented in RQ1 should be interpreted with caution, as our findings are based on a relatively small benchmark and are not conclusive. While they offer useful guidance for future investigations, they should not be taken as definitive evidence that one prompt strategy universally outperforms others. The purpose of these experiments was to identify a prompt engineering configuration that was "good enough" to reliably generate functional tests and enable integration with search-based testing.

Furthermore, the strategies explored represent only a small number of possible approaches. The literature review that informed our selection lists over 15 distinct prompting techniques. Even within the strategies we tested, there remains room for fine-tuning and further optimization. Future work should revisit this space with larger benchmarks and additional evaluation criteria.

Moving on to our integration setup, a key limitation here is the difficulty in comparing the two approaches fairly. LLM queries were executed in the cloud on proprietary infrastructure with access to powerful pre-trained models, whereas SynTest ran locally from scratch. These differences mean the two approaches are not directly comparable in terms of compute budget or data priors. It is possible that SynTest would achieve higher coverage if it was run under the same circumstances.

Finally, our similarity analysis, used to estimate whether LLM-generated tests are novel or memorized, has important limitations with respect to potential training data leakage. While approximately 70% of the generated tests were classified as non-similar to the project's own manual test suite, our comparison is limited to those specific test suites. It remains entirely possible that similar tests exist in other open-source repositories that were part of the LLM's training data. Since we do not have access to the training set used by the model, we cannot rule out this possibility. Therefore, while our results suggest that LLMs are often capable of generalizing, we cannot definitively exclude memorization from other unseen sources.

6.5. Threats to Validity

Internal Validity — While significant time and effort was invested to ensure the correctness of our implementation, certain implementation errors still occurred. For example, the issue discovered in the nQueens.js class, where runtime errors prevented proper test execution. Such bugs can affect the reliability of some experimental outcomes.

External Validity — A major concern is data leakage, as our benchmarks are almost guaranteed to be included in the training datasets used by the evaluated LLMs. This means our LLM-based results may be overly optimistic and not representative of their effectiveness on unseen code. We did not take explicit measures to mitigate this concern due to the time- and resources required for creating a new benchmark. Future research could address this threat by evaluating on proprietary or modified benchmarks to ensure more realistic generalization.

Conclusion Validity — Our experiments involve inherently non-deterministic methods (both LLM prompting and search-based evolutionary algorithms). To mitigate randomness , each experiment was repeated ten times. Furthermore, we based our conclusions off sound statistical significance testing. However, conclusions must be interpreted cautiously, given the possibility of data leakage influencing LLM results. Future work should further validate these findings in scenarios with minimized data leakage.

7 Conclusion

In this thesis, we introduced LLM-Seeded Evolutionary Testing (LSET), a hybrid approach combining the strengths of Search-Based Software Testing (SBST) and Large Language Models (LLMs). To address the limitation of SBST struggling with generating test cases with complex method sequences, we utilized the natural language capabilities of LLMs to generate semantically coherent test cases to be used as seeds.

Through a series of experiments, this work evaluated multiple strategies for prompting LLMs to generate high-quality test cases. The most suitable strategy was Self-Refine, where the model is prompted multiple times to reflect and refine its produced output. This approach was the most consistent in generating complex method sequences with the trade-off of being the most computationally expensive.

After obtaining the test cases, we incorporated them into the SBST process by utilizing them as the seeds for the initial population. Empirical results showed that LSET improved SBST test structure and achieved superior branch coverage compared to standalone SBST or LLMs alone.

However, there was high variance in the quality of structure within the final LSET test suites, ranging from carbon copies of LLM tests to tests without a trace of LLM influence, with hybrid cases in between these extremes. We found that LSET has the potential to create test cases which utilize the structure provided by LLMs to uncover new branches and retain readability. However, this was not guaranteed as there are many hybrid test cases with redundant code fragments that do not meaningfully use the original seeds.

While LSET outperformed individual approaches in coverage, its advantages diminished when compared to simpler integration strategies. When evaluated against the straightforward combination of SynTest and LLM test suites, LSET only achieved superior coverage in 1 out of 35 classes, with the rest showing no statistically significant difference.

This modest improvement is due to various factors including limitations in parsing, occasional instability in LLM output, and constraints in the benchmark. However, the most critical limitation is that LSET is only as effective as the quality of its seeds, if they fail to provide the proper foundation the search process cannot build upon them effectively and uncover key branches.

Despite the marginal results, LSET still has much potential and multiple potential directions for future work. To preserve structure, we can extract the complex method sequences from the LLM-generated test cases and leverage them throughout the evolutionary process. In order to potentially improve coverage, we can look into dynamic prompting where the LLM is steered toward uncovered branches by including previous results in follow-up prompts. Furthermore, the individual components of LSET can be refined by investigating more prompt engineering strategies, and making parser improvements. Nevertheless, for scenarios where semantic structure are prioritized over pure coverage metrics, LSET offers a promising direction for automated test generation.

In conclusion, LSET represents a novel approach to enhancing automated software testing by leveraging the combined strengths of SBST and LLMs, contributing to the growing field of hybrid automated testing methodologies. While current limitations present challenges, particularly regarding seed quality and the consistency of improvements, the foundational potential of LSET remains clear. Our findings provide important insights into both the opportunities and challenges of integrating search-based testing with language models, advancing understanding of how different automated testing techniques can be effectively combined. Future research, focusing on targeted structure preservation and dynamic prompting, alongside continuous refinement of LSET's core components, will help LSET evolve toward its full potential for generating more effective and efficient software tests.

Bibliography

[1] Raja Ben Abdessalem, Annibale Panichella, Shiva Nejati, Lionel C Briand, and Thomas Stifter. "Testing autonomous cars for feature interaction failures using many-objective search". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 2018, pp. 143–154.

[2] Azat Abdullin, Pouria Derakhshanfar, and Annibale Panichella. "Test Wars: A Comparative Study of SBST, Symbolic Execution, and LLM-Based Approaches to Unit Test Generation". In: *arXiv preprint arXiv:2501.10200* (2025).

[3] Shaukat Ali, Lionel C Briand, Hadi Hemmati, and Rajwinder Kaur Panesar-Walawege. "A systematic review of the application and empirical investigation of search-based test case generation". In: *IEEE Transactions on Software Engineering* 36.6 (2009), pp. 742–762.

[4] M Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. "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. 2017, pp. 263–272.

[5] Andrea Arcuri and Gordon Fraser. "Parameter tuning or default values? An empirical investigation in search-based software engineering". In: *Empirical Software Engineering* 18 (2013), pp. 594–623.

[6] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. "When, how, and why developers (do not) test in their IDEs". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 2015, pp. 179–190.

[7] José Campos, Yan Ge, Nasser Albunian, Gordon Fraser, Marcelo Eler, and Andrea Arcuri. "An empirical evaluation of evolutionary algorithms for unit test suite generation". In: *Information and Software Technology* 104 (2018), pp. 207–235.

[8] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. "Chatunitest: A framework for llm-based test generation". In: *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 2024, pp. 572–576.

[9] Shyam R Chidamber and Chris F Kemerer. "Towards a metrics suite for object oriented design". In: *Conference proceedings on Object-oriented programming systems, languages, and applications*. 1991, pp. 197–211.

[10] Ermira Daka, José Campos, Gordon Fraser, Jonathan Dorn, and Westley Weimer. "Modeling readability to improve unit tests". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 2015, pp. 107–118.

[11] Khalid El Haji, Carolin Brandt, and Andy Zaidman. "Using GitHub Copilot for Test Generation in Python: An Empirical Study". In: *Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024)*. 2024, pp. 45–55.

[12] Gordon Fraser and Andrea Arcuri. "1600 faults in 100 projects: automatically finding faults while achieving high coverage with evosuite". In: *Empirical software engineering* 20 (2015), pp. 611–639.

[13] Gordon Fraser and Andrea Arcuri. "The seed is strong: Seeding strategies in search-based software testing". In: *2012 IEEE fifth international conference on software testing, verification and validation*. IEEE. 2012, pp. 121–130.

[14] Daya Guo, Canwen Xu, Nan Duan, Jian Yin, and Julian McAuley. "Longcoder: A long-range pre-trained language model for code completion". In: *International Conference on Machine Learning*. PMLR. 2023, pp. 12098–12107.

[15] Mark Harman, Yue Jia, and Yuanyuan Zhang. “Achievements, open problems and challenges for search based software testing”. In: *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2015, pp. 1–12.

[16] Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. “Self-planning code generation with large language models”. In: *ACM Transactions on Software Engineering and Methodology* 33.7 (2024), pp. 1–30.

[17] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. “Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models”. In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE. 2023, pp. 919–931.

[18] Chao Ran Erwin Li. *Replication Package for ”LLM-Seeded Evolutionary Testing (LSET): Enhancing Search-Based Software Testing by Incorporating Complex Method Sequences”*. Version associated with Master’s thesis at Delft University of Technology. 2025. DOI: 10.5281/zenodo.15625223. URL: https://doi.org/10.5281/zenodo.15625223.

[19] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. “Structured chain-of-thought prompting for code generation”. In: *ACM Transactions on Software Engineering and Methodology* 34.2 (2025), pp. 1–23.

[20] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. “Self-refine: Iterative refinement with self-feedback”. In: *Advances in Neural Information Processing Systems* 36 (2023), pp. 46534–46594.

[21] Ggaliwango Marvin, Nakayiza Hellen, Daudi Jjingo, and Joyce Nakatumba-Nabende. “Prompt engineering in large language models”. In: *International conference on data intelligence and cognitive informatics*. Springer. 2023, pp. 387–402.

[22] Matteo Modonato. “Combining Dynamic Symbolic Execution, Machine Learning and Search-Based Testing to Automatically Generate Test Cases for Classes”. In: *arXiv preprint arXiv:2005.09317* (2020).

[23] Gene Myers. “A fast bit-vector algorithm for approximate string matching based on dynamic programming”. In: *Journal of the ACM (JACM)* 46.3 (1999), pp. 395–415.

[24] Wendkûuni C Ouédraogo, Kader Kaboré, Haoye Tian, Yewei Song, Anil Koyuncu, Jacques Klein, David Lo, and Tegawendé F Bissyandé. “Large-scale, Independent and Comprehensive study of the power of LLMs for test case generation”. In: *arXiv preprint arXiv:2407.00225* (2024).

[25] Shuyin Ouyang, Jie M Zhang, Mark Harman, and Meng Wang. “LLM is Like a Box of Chocolates: the Non-determinism of ChatGPT in Code Generation”. In: *arXiv preprint arXiv:2308.02828* (2023).

[26] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. “Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets”. In: *IEEE Transactions on Software Engineering* 44.2 (2017), pp. 122–158.

[27] Sebastiano Panichella, Alessio Gambi, Fiorella Zampetti, and Vincenzo Riccio. “Sbst tool competition 2021”. In: *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*. IEEE. 2021, pp. 20–27.

[28] Martin Riddell, Ansong Ni, and Arman Cohan. “Quantifying contamination in evaluating code generation capabilities of language models”. In: *arXiv preprint arXiv:2403.04811* (2024).

[29] José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. “Seeding strategies in search-based unit test generation”. In: *Software Testing, Verification and Reliability* 26.5 (2016), pp. 366–401.

[30] Pranab Sahoo, Ayush Kumar Singh, Sriparna Saha, Vinija Jain, Samrat Mondal, and Aman Chadha. “A systematic survey of prompt engineering in large language models: Techniques and applications”. In: *arXiv preprint arXiv:2402.07927* (2024).

[31] Martina Saletta and Claudio Ferretti. “Exploring the prompt space of large language models through evolutionary sampling”. In: *Proceedings of the Genetic and Evolutionary Computation Conference*. 2024, pp. 1345–1353.

[32] Arkadii Sapozhnikov, Mitchell Olsthoorn, Annibale Panichella, Vladimir Kovalenko, and Pouria Derakhshanfar. "TestSpark: IntelliJ IDEA's Ultimate Test Generation Companion". In: *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*. 2024, pp. 30–34.

[33] Rylan Schaeffer. "Pretraining on the test set is all you need". In: *arXiv preprint arXiv:2309.08632* (2023).

[34] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. "An empirical evaluation of using large language models for automated unit test generation". In: *IEEE Transactions on Software Engineering* 50.1 (2023), pp. 85–105.

[35] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. "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. 2015, pp. 201–211.

[36] Mohammed Latif Siddiq, Joanna Cecilia Da Silva Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinícius Carvalho Lopes. "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*. 2024, pp. 313–322.

[37] Stack Overflow. *Stack Overflow Developer Survey 2024*. https://survey.stackoverflow.co/2024/technology. Accessed: April 20, 2025. 2024.

[38] Dimitri Stallenberg, Mitchell Olsthoorn, and Annibale Panichella. "Guess what: Test case generation for Javascript with unsupervised probabilistic type inference". In: *International Symposium on Search Based Software Engineering*. Springer. 2022, pp. 67–82.

[39] Philipp Straubinger and Gordon Fraser. "A Survey on What Developers Think About Testing". In: *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE. 2023, pp. 80–90.

[40] Yutian Tang, Zhijie Liu, Zhichao Zhou, and Xiapu Luo. "Chatgpt vs sbst: A comparative assessment of unit test suite generation". In: *IEEE Transactions on Software Engineering* (2024).

[41] Paolo Tonella. "Evolutionary testing of classes". In: *ACM SIGSOFT Software Engineering Notes* 29.4 (2004), pp. 119–128.

[42] Catherine Tony, Nicolás E Díaz Ferreyra, Markus Mutas, Salem Dhiff, and Riccardo Scandariato. "Prompting Techniques for Secure Code Generation: A Systematic Investigation". In: *arXiv preprint arXiv:2407.07064* (2024).

[43] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. "Unit test case generation with transformers and focal context". In: *arXiv preprint arXiv:2009.05617* (2020).

[44] Lukas Twist, Jie M Zhang, Mark Harman, Don Syme, Joost Noppen, and Detlef Nauck. "LLMs Love Python: A Study of LLMs' Bias for Programming Languages and Libraries". In: *arXiv preprint arXiv:2503.17181* (2025).

[45] Shubham Ugare, Tarun Suresh, Hangoo Kang, Sasa Misailovic, and Gagandeep Singh. "SynCode: LLM generation with grammar augmentation". In: *arXiv preprint arXiv:2403.01632* (2024).

[46] András Vargha and Harold D Delaney. "A critique and improvement of the CL common language effect size statistics of McGraw and Wong". In: *Journal of Educational and Behavioral Statistics* 25.2 (2000), pp. 101–132.

[47] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. "Software testing with large language models: Survey, landscape, and vision". In: *IEEE Transactions on Software Engineering* (2024).

[48] Richard A Watson, Gregory S Hornby, and Jordan B Pollack. "Modeling building-block interdependency". In: *Parallel Problem Solving from Nature—PPSN V: 5th International Conference Amsterdam, The Netherlands September 27–30, 1998 Proceedings 5*. Springer. 1998, pp. 97–106.

[49] Richard A Watson and Thomas Jansen. “A building-block royal road where crossover is provably essential”. In: *Proceedings of the 9th annual conference on Genetic and evolutionary computation*. 2007, pp. 1452–1459.

[50] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. “Chain-of-thought prompting elicits reasoning in large language models”. In: *Advances in neural information processing systems* 35 (2022), pp. 24824–24837.

[51] CONOVER WJ. *“Practical nonparametric statistics. vol. 350*. 1998.

[52] Chen Yang, Junjie Chen, Bin Lin, Jianyi Zhou, and Ziqi Wang. “Enhancing llm-based test generation for hard-to-cover branches via program analysis”. In: *arXiv preprint arXiv:2404.04966* (2024).

[53] Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. “Evaluating and improving chatgpt for unit test generation”. In: *Proceedings of the ACM on Software Engineering* 1.FSE (2024), pp. 1703–1726.

[54] Xunyu Zhu, Jian Li, Yong Liu, Can Ma, and Weiping Wang. “A survey on model compression for large language models”. In: *Transactions of the Association for Computational Linguistics* 12 (2024), pp. 1556–1577.

A

Prompt Templates

```
--- Zero-Shot ---
[INST]
You are an expert Javascript tester.
Your goal is to write test cases that are readable and understandable to other
developers while maximizing coverage of the class-under-test.
Include test cases with multiple method calls where relevant.
These involve sequences of method invocations on the same object, where one method's
    behavior or state affects others.
Use such tests when the 'class methods share state, have side effects, or depend on
    logical sequencing.
Avoid unnecessary complexity: if the methods are independent or do not benefit from
    sequencing, use standalone tests instead.
Place the final test suite between the [OUTPUT] and [/OUTPUT] tags.
The class-under-test is provided between the [CODE] and [/CODE] tags.
Do not return any text outside the [OUTPUT] and [/OUTPUT] tags.
[/INST]
[CODE] class_code [/CODE]
[OUTPUT] [/OUTPUT]
```

```
--- Chain-of-Thought ---
[INST] You are an expert JavaScript tester. Your task is to generate test cases for the
     class-under-test. Your goal is to write test cases that are readable and
    understandable to other developers while ensuring complete coverage of all
    functionalities, methods, and edge cases of the class. Please follow these
    instructions:

1. Carefully review the class-under-test provided between the [CODE] and [/CODE] tags.
2. Think and explain your reasoning step by step before writing the test cases:
   - Identify all methods and properties of the class.
   - Consider their expected behaviors.
   - Think through potential edge cases and boundary conditions.
   - Decide on the best test strategy to ensure full coverage.
   -Include test cases with multiple method calls where relevant.
    These involve sequences of method invocations on the same object, where one methods
         behavior or state affects others.
    -Use such tests when the class's methods share state, have side effects, or depend
        on logical sequencing.
    -Avoid unnecessary complexity: if the methods are independent or do not benefit
        from sequencing, use standalone tests instead.
3. Place only the final test code between the [OUTPUT] and [/OUTPUT] tags.
4. Do not include any text outside the [OUTPUT] and [/OUTPUT] tags.

[CODE]
class_code
[/CODE]
[OUTPUT]
[/OUTPUT]
```

```
--- Self-Refine Initial ---
[INST]
You are an expert Javascript tester.
Your goal is to write test cases that are readable and understandable to other
developers while maximizing coverage of the class-under-test.
Include test cases with multiple method calls where relevant.
These involve sequences of method invocations on the same object, where one method's
    behavior or state affects others.
Use such tests when the 'class methods share state, have side effects, or depend on
    logical sequencing.
Avoid unnecessary complexity: if the methods are independent or do not benefit from
    sequencing, use standalone tests instead.
Place the final test suite between the [OUTPUT] and [/OUTPUT] tags.
The class-under-test is provided between the [CODE] and [/CODE] tags.
Do not return any text outside the [OUTPUT] and [/OUTPUT] tags.
[/INST]
[CODE] class_code [/CODE]
[OUTPUT] [/OUTPUT]
```

```
--- Self-Refine Reflection ---
[INST]
Here is a test suite: self_refine_initial.
Please review the test suite and suggest any improvements or identify any missing edge
    cases or issues.
Provide feedback in terms of readability, coverage, and structure of the tests.
Specifically, check whether the test suite includes tests with multiple method calls
    where relevant.
These involve sequences of method invocations on the same object, where one method's
    behavior or state affects others
Place the feedback between the [OUTPUT] and [/OUTPUT] tags.
Do not return any text outside the [OUTPUT] and [/OUTPUT] tags
[/INST]
[OUTPUT]
[/OUTPUT]
```

```
--- Self-Refine Refinement ---
[INST]
Here is the test suite to be improved  self_refine_initial.
Based on the following feedback, refine the test suite to include the suggested
    improvements:self_refine_reflection.
Ensure that the final test suite is comprehensive, readable, and well-structured,
    covering all the important scenarios for the class-under-test,including tests with
     multiple method calls where relevant
Place the final test suite between the [OUTPUT] and [/OUTPUT] tags.
Do not return any text outside the [OUTPUT] and [/OUTPUT] tags.
[/INST]
[OUTPUT]
[/OUTPUT]
```

```
--- Self-Planning Plan ---
[INST]
You are an expert JavaScript tester.
Review the provided class and create a structured testing plan like in the examples.
The plan should cover all essential scenarios and boundary conditions, including test
    cases with multiple method calls where relevant.
These involve sequences of method invocations on the same object, where one method's
    behavior or state affects others
Place the plan between [OUTPUT] and [/OUTPUT] tags, and use the class code given
    between [CODE] and [/CODE] tags.
Do not return any text outside the [OUTPUT] and [/OUTPUT] tags.
[/INST]

[EXAMPLE]
class Calculator {
  add(a, b) {{ return a + b; }
  subtract(a, b) { return a - b; }
  divide(a, b) { return a / b; }
}}
```

```
Plan:
1. Test addition with positive and negative numbers.
2. Test subtraction with zero, positive, and negative numbers.
3. Test division by non-zero values and handle division by zero.
4. Test large and small input values for potential overflow.
[/EXAMPLE]

[CODE]
class_code
[/CODE]
[OUTPUT]
[/OUTPUT]
```

```
--- Self-Planning Implementation ---
[INST]
Here is a detailed plan: self_planning_plan.
Based on this plan, implement JavaScript test cases that maximizes coverage and are
    easy to understand for other developers.
Ensure that the test cases include tests with multiple method calls where relevant, as
    per the plan.
Place the final test cases between the [OUTPUT] and [/OUTPUT] tags, and use the class
    code given between [CODE] and [/CODE] tags.
Do not return any text outside the [OUTPUT] and [/OUTPUT] tags.
[/INST]
[CODE]
class_code
[/CODE]
[OUTPUT]
[/OUTPUT]
```