The Status of JavaScript Test Generation: A Benchmark-Based Evaluation

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Saga Rut Sunnevudóttir

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

JetBrains
Gelrestraat 16
Amsterdam, the Netherlands
www.jetbrains.com

The Status of JavaScript Test Generation: A Benchmark-Based Evaluation

Abstract

Automated test generation is a critical area of research in software engineering, aiming to reduce manual effort while improving software reliability. While substantial work has focused on statically typed languages, dynamically typed languages such as JavaScript remain underexplored despite their widespread use and unique challenges. This thesis investigates the current status of JavaScript test generation by systematically evaluating state-of-the-art search-based and large language model-based tools.

We first analyze existing benchmarks to assess their coverage of representative language features, identifying gaps that limit the ability to fairly compare tool performance. We then construct a curated dataset of real-world JavaScript projects and evaluate the LLM-based tool TestPilot and the search-based tool SynTest using a combination of quantitative metrics (e.g., code coverage, pass rates) and feature based correlation analysis. Our results reveal that TestPilot tends to generate higher coverage (median 27.9% vs 11.2% branch coverage) and more readable tests but produces a larger number of failing or low-value test cases, while SynTest generates more stable and focused test suites yet can struggle with complex or dynamic code constructs. Our similarity analysis shows that each approach achieved unique coverage, suggesting complementary strengths.

This study highlights the need for standardized, language-aware benchmarks and introduces a curated dataset and evaluation framework for evaluating JavaScript test generation tools. By systematically comparing search-based and LLM-based approaches, this thesis offers insights into their respective strengths, limitations, and opportunities for hybrid strategies, advancing the state of automated testing for dynamically typed languages.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. M.J.G. Olsthoorn, Faculty EEMCS, TU Delft
Company supervisor:	Dr. P. Derakhshanfar, JetBrains
External Committee Member:	Dr. A. Costea, Faculty EEMCS, TU Delft

Preface

As I finish both this project and my time in Delft, I would like to thank those who have supported me along the way. First of all, I would like to thank my supervisors, Mitchell and Pouria, for their feedback, guidance, and continued support throughout the thesis. I am also grateful to Arie, my thesis advisor, for his insightful suggestions, and to Andreea for serving on my thesis committee. In addition, I want to thank everyone involved in the AI4SE collaboration between TU Delft and JetBrains for providing me with this valuable opportunity and inspiring me to continue in research.

I want to thank my friends back home for always being a welcome distraction when needed and my people in Delft, Zoë, Abhi, and Aykut, for all the good times making Delft our second hometown, from the very first week til the last.

Last but not least, I want to thank my family, especially my grandma, who has been by my side through every academic achievement, as much now as ever.

Thank you!

Saga Rut Sunnevudóttir
Delft, the Netherlands
August 15, 2025

Contents

List of Figures

Chapter 1

Introduction

Software testing remains a critical yet challenging aspect of software engineering. It is often time-consuming, costly, and prone to human error especially as software systems grow more complex [42]. For this reason, automated test case generation has become a highly researched solution enabling developers to create test cases more efficiently, reducing manual labor while maintaining high software quality [66] [60]. Among these approaches, search-based software testing (SBST) has proven effective, with studies showing that it can generate high-coverage test suites and achieve high fault detection [22] [48]. However, SBST-generated tests often lack readability, can be hard to maintain, and may get trapped in local optima during the search process [18] [33] [49]. To address these limitations, recent research has explored Large Language Model (LLM)-based approaches, which not only achieve high coverage but also produce more readable tests, capture intricate edge cases, and generate natural language-based test inputs [77] [16] [68]. Test generation techniques vary widely depending on the programming language's characteristics, with different methods often tailored to specific programming languages. Dynamically typed languages (DTLs), such as JavaScript, are among the most popular programming languages today [1] [2]. However, their flexible type systems and runtime behaviors introduce distinct challenges for automated test generation [37]. Despite their popularity, they are underexplored in comparison to statically typed languages. Given their widespread usage and the lack of proper benchmarking, it is essential to systematically evaluate and compare state-of-the-art test generation tools to assess their effectiveness for these languages.

1.1 Problem Definition

Despite growing interest in automated test generation for JavaScript, there is a lack of standardized and representative benchmarks to evaluate and compare existing tools, compared to other programming languages [30] [72] [65] [15] [31] [63] [21]. Tools are often assessed on handpicked or popular projects without systematic selection criteria, raising concerns about dataset bias, metric inconsistency, and result reproducibility [67] [56] [39].

This is particularly problematic for JavaScript, a DTL that lacks native type annotations and introduces unique challenges such as prototype-based inheritance, asynchronous

execution, and dynamic object manipulation [10] [7] [17]. These features complicate the inference of expected behavior and increase the difficulty of generating meaningful test cases.

Moreover, benchmarking remains an often overlooked aspect of test generation research [21] [48]. Current evaluations frequently rely on coverage metrics alone (e.g., statement or branch coverage), offering little insight into why tools perform as they do or what language aspects are adequately tested [67] [56]. The lack of language-level evaluation and a unified evaluation framework makes a fair, insightful comparison between techniques, especially search-based and LLM-based approaches, difficult [4].

Thus, the core problem addressed in this thesis is the absence of robust, language-aware benchmarks and standardized evaluation practices for JavaScript test generation. Without these, it is difficult to compare tools fairly, assess their limitations, or improve their generalizability. To address this, we evaluate existing benchmarks in terms of language feature representation and propose a framework for assessing both datasets and tool performance.

1.2 Research aim and scope

In this study, we aim to:

- Investigate how existing JavaScript test generation tool benchmarks cover unique language features relevant to test generation.
- Measure how current SOTA JavaScript test generation tools perform.
- Analyze potential reasons for tool performance variations based on code complexity and language feature diversity.

Our hypothesis is that tools will potentially underperform on projects with complexities or language features that are not represented in the benchmarks they were developed and evaluated on. By addressing these aims, we seek to provide a deeper understanding of the strengths and limitations of current tools and benchmarks, and to inform future improvements in both areas. The specific research questions that follow from these research aims are listed in Chapter 5.

Research on JavaScript testing often differentiates between front-end and back-end testing. While covering all aspects of testing is challenging, the generalizability of LLM-based tools gives them the potential to address a broader scope. In contrast, search-based tools tend to focus primarily on one aspect, either front-end or back-end testing [34] [41] [67]. As a result, for fair comparison, the focus in this study will be on back-end testing in JavaScript.

1.3 Research Contributions

In this thesis, we investigate the effectiveness of automated test generation tools for JavaScript by developing a comprehensive benchmarking framework. Our contributions can be summarized as follows:

- **Evaluation of Existing Benchmarks** We analyze existing test generation benchmarks on JavaScript language feature coverage and diversity to assess whether they are suitable for fully evaluating the performance and generalizability of test generation tools.
- **Benchmark** We construct a well-curated dataset of JavaScript projects for evaluating test generation tools comprised of diverse, representative and complex projects.
- **Evaluation of Existing Tools** We benchmark both search-based and LLM-based test generation tools on the same dataset, assessing them in terms of code coverage and feature-based strengths/weaknesses. This evaluation enables tool improvement and further research in this field.

1.4 Thesis Outline

This thesis is structured as follows. Chapter 2 introduces the background on software testing, automated test generation, benchmarking and the unique characteristics of JavaScript. Chapter 3 reviews related work, focusing on both search-based and LLM-based test generation techniques, as well as existing comparative studies and highlights the current research gap. Chapter 4 describes the design and construction of the benchmark dataset, including the criteria for project selection and feature analysis. Chapter 5 explains the experimental setup and evaluation procedures. Chapter 6 presents and analyzes the results of the comparative evaluation. Chapter 7 offers a discussion of the findings and their threats to validity. Finally, Chapter 8 concludes the thesis and outlines directions for future research.

Chapter 2

Background

This section provides an overview of key concepts relevant to this research, including software testing challenges, automated test generation techniques, characteristics of JavaScript and common evaluation metrics for test generation tools. This foundation will help contextualize the analysis and benchmarking of JavaScript test generation tools discussed later in this thesis.

2.1 Software Testing

Software testing is a fundamental practice in software development to ensure the correctness, reliability and maintainability of software systems. By executing code under different conditions, testing helps detect defects early in the development process, reducing the risk of failures in production [42] [5]. Understanding software testing concepts is essential for contextualizing automated test generation techniques, which are the primary focus of this thesis. The following subsections introduce the main types of software testing, the metrics and quality considerations used to evaluate test suites, and common challenges that automated test generation aims to address.

2.1.1 Types of Software Testing

Software testing can be categorized into different levels based on the scope and purpose of the tests [69]:

- **Unit Testing:** Focuses on testing individual components or functions in isolation. Unit tests verify that small, self-contained pieces of code behave as expected.
- **Integration Testing:** Ensures that different components or modules work together correctly. It verifies interactions between services, databases or APIs.
- **System Testing:** Validates the behavior of the complete system to ensure it meets specified technical requirements. It tests the system as a whole in an environment that closely mirrors production.

- **Acceptance Testing:** Verifies whether the system meets the business requirements and is ready for deployment. Typically performed from the end-user or stakeholder perspective to ensure the system fulfills its intended use.

2.1.2 Goals and Metrics in Software Testing

Several metrics are commonly used to evaluate a test suite [24]:

- **Code Coverage:** Measures the proportion of code executed and tested by the test suite (e.g. statement, branch and path coverage).
- **Mutation Score:** Evaluates the ability of tests to detect artificially injected faults.
- **Fault Detection Rate:** Assesses how effectively the test suite uncovers real defects in the software.

In addition to these quantitative metrics, the quality of test cases is crucial. Poor test quality can introduce test smells, which are indicators of problematic tests that may reduce reliability, readability, or maintainability. Examples include:

- *Mystery Guest:* Tests that interact with external resources such as files or databases, making them non-deterministic and harder to isolate.
- *Eager Test:* Tests that validate multiple functionalities in a single case, reducing readability and increasing debugging difficulty.
- *Assertion Roulette:* Tests with multiple assertions lacking clear failure messages, making it unclear which assertion caused a failure.

By combining quantitative metrics with attention to test smells, developers can improve both the effectiveness and the maintainability of their test suites [50].

2.1.3 Challenges in Software Testing

Despite its advances, software testing faces several challenges. Writing comprehensive test suites is time-consuming and requires deep knowledge of the system under test, often leading to gaps in coverage [42]. Large and complex codebases make it difficult to systematically generate test cases that explore all possible execution paths [6] [32]. Ensuring sufficient input diversity is another challenge, as manually crafted tests may not account for uncommon edge cases or unexpected user behaviors [32]. Furthermore, as software systems evolve, existing tests may become outdated or need to be modified, increasing maintenance overhead [64]. Addressing these challenges is essential for improving the efficiency, effectiveness, and reliability of software testing.

Automated test generation aims to address these challenges by producing tests systematically and efficiently. Understanding these goals, metrics, and challenges provides the necessary background for the analysis and benchmarking of automated test generation tools discussed later in this thesis.

2.2 Automated Test Generation

Automated test generation employs a variety of techniques to reduce manual effort and produce diverse, high-quality test cases [6]. This section discusses different approaches, including fuzzing, symbolic execution and concolic testing, search-based software testing (SBST), and machine learning (ML) or large language model (LLM)-based software testing, highlighting their strengths and limitations.

2.2.1 Fuzzing

Fuzzing is an automated testing technique that generates and mutates inputs to expose unexpected behaviors, such as crashes and assertion failures [35]. Its random nature allows it to uncover non-trivial edge cases and security vulnerabilities. It is highly scalable and can quickly test a wide range of inputs, making it effective at finding low-level bugs. While simple fuzzers rely on random generation, more advanced ones like AFL use coverage-guided strategies to systematically explore input spaces [79]. However, fuzzing lacks semantic awareness, often generating irrelevant inputs that fail to explore deeper execution paths.

2.2.2 Symbolic Execution & Concolic Testing

Symbolic execution explores program paths by treating inputs as symbolic variables, generating constraints to analyze execution behavior [6]. It offers high code coverage and precise test case generation, making it effective for identifying assertion failures and exceptions. Concolic testing improves symbolic execution by using concrete inputs to increase code coverage [59]. Tools such as KEX [3] and UnitTestBot [28] exemplify this approach, generating input values that exercise diverse execution paths and often achieving high branch coverage. However, both methods face the issue of path explosion, where the number of execution paths grows rapidly, limiting their scalability.

2.2.3 Search-Based Software Testing (SBST)

SBST formulates test generation as an optimization problem, using techniques like genetic algorithms to evolve test cases toward specific goals, such as maximizing code coverage, often with branch coverage as the fitness function [6]. This approach provides guided exploration of the search space and can thus handle complex input structures and achieve high coverage [21] [48] [37] [49]. Notable tools implementing these techniques include EvoSuite for Java [21] and Pynguin for Python [37]. However, evolutionary algorithms have high computational costs due to the need for multiple test executions, and they may struggle with local optima, where the search plateaus before finding optimal test cases [33] [49]. Additionally, the generated tests can lack readability, making it harder to interpret and maintain them [18].

2.2.4 Machine Learning and LLM-Based Software Testing

Machine learning and LLMs use data-driven approaches to generate tests, learning patterns from software repositories [29] [56] [33]. Compared to search-based techniques, LLMs have more generalizability, produce more meaningful assertions, improve test readability and can capture complex edge cases, especially with string inputs [4] [16] [68]. Tools such as TestPilot [56], ChatUniTest [16], AgoneTest [36], and TestSpark [55] exemplify LLM-based test generation. However, their effectiveness depends on training data quality, leading to issues like hallucinations and data leakage [29] [4]. While LLM-based testing reduces human effort, it may offer lower coverage and lacks explainability, making it harder to trust the generated test cases.

2.3 JavaScript

JavaScript is a widely used programming language, particularly in web development. Initially created in 1995 as a scripting language for web browsers, JavasScript has evolved into a full-fledged programming language used in both frontend and backend development, and is one of the most popular languages today [2] [1].

2.3.1 Characteristics and Features

Key features of JavaScript include first-class functions, dynamic typing, asynchronous programming with promises and async/await, event-driven programming, and prototype-based inheritance [17]. These features provide flexibility and ease of use but also introduce challenges in debugging and maintainability due to their dynamic nature and sometimes unpredictable behavior.

2.3.2 Testing Methods and Frameworks

JavaScript supports an ecosystem of libraries and frameworks that streamline development. Popular frontend frameworks like React [1], Angular [2], and Vue.js [3] offer component-based architectures, while backend frameworks such as Node.js [4], Express [5], and NextJS [6] provide server-side solutions.

The type of testing required often depends on the framework or library used. For example, React applications focus on component testing with Jest [7] and React Testing Library [8], while backend frameworks like Express may emphasize API and integration testing.

[1] https://react.dev/
[2] https://angular.dev/
[3] https://vuejs.org/
[4] https://nodejs.org/en
[5] https://expressjs.com/
[6] https://nextjs.org/
[7] https://jestjs.io/
[8] https://testing-library.com/docs/react-testing-library/intro/

Several frameworks facilitate testing in JavaScript. Jest is widely used for unit and snapshot testing with built-in mocking, while Mocha [9] and Chai [10] offer flexibility for unit and integration tests. For end-to-end testing, Cypress [11] and Playwright [12] simulate real user interactions, though the approach varies depending on the framework.

2.4 Evaluation and Benchmarking in Test Generation

Evaluating test generation techniques requires systematic benchmarking to compare their effectiveness, efficiency, and applicability across different software projects. Benchmarks provide standardized datasets and testing scenarios, enabling fair comparisons between test generation tools. This section discusses the process of creating benchmarks, their role in software testing research, the different types of benchmarks, and the key evaluation criteria used to assess test generation approaches.

2.4.1 Benchmarks

Benchmarks serve as controlled environments for evaluating the performance of test generation tools. The choice of benchmark significantly impacts the extent to which the evaluation reflects real-world performance, as it determines how well a test generation approach performs across diverse software projects [21].

A well-designed benchmark must accurately test the tools, techniques, and algorithms under evaluation, ensuring it is both representative of real-world software and the programming language's characteristics [78]. This requires selecting complex and diverse projects that capture real-world usage patterns. Benchmarks for test generation can be split into two types: general projects [30] [73] [65] and those specifically comprised of buggy code to evaluate fault detection capabilities [31] [63] [74] [27]. Both types are useful, as they provide insights into how tools perform in typical development environments or in scenarios where the goal is to identify and address software defects.

To ensure fairness, benchmarks must be unbiased, avoiding cherry-picked projects that could favor a specific tool or produce unrealistic results. One common way to achieve this is by collecting open-source projects from platforms like GitHub, which can provide a broad and realistic sample [56] [29]. Ultimately, well-designed benchmarks are essential for producing meaningful and comparable evaluations of test generation tools, guiding the development of more effective approaches.

2.4.2 Evaluation Criteria

To assess the effectiveness of test generation tools, various evaluation criteria and metrics are used. One of the most fundamental metrics is code coverage, which measures the extent of which generated tests execute different parts of the program, including statement, branch,

[9] https://mochajs.org/
[10] https://www.chaijs.com/
[11] https://www.cypress.io/
[12] https://playwright.dev/

and path coverage [24]. Higher coverage indicates better test effectiveness, but it does not necessarily guarantee fault detection [24].

Another important criteria is fault detection capability, which evaluates how well generated tests find bugs in software. Mutation testing, where artificial faults are introduced into the program, is commonly used to measure this capability [52].

Test suite size and redundancy are also considered when evaluating test generation tools. A test suite should be minimal yet effective, avoiding excessive or redundant test cases that do not contribute to additional coverage [44].

Beyond quantitative metrics, qualitative aspects such as readability and maintainability of generated tests are important. Tests that are difficult to understand or maintain may be of limited practical use, even if they achieve high coverage [12].

By combining well-designed benchmarks with appropriate quantitative and qualitative evaluation criteria, researchers can comprehensively assess test generation tools. These evaluations reveal not only how effectively a tool explores code and detects faults, but also how usable and maintainable its generated tests are. In this thesis, these principles guide the design of the benchmark and selection of metrics used to systematically compare SBST and LLM-based test generation tools for JavaScript, ensuring that the analysis reflects both technical effectiveness and practical utility.

Chapter 3

Related Work

Automated test generation has been extensively studied across different programming languages, with research focusing on both search-based and machine-learning approaches. This chapter provides an overview of prior work in the field covering different languages and test generation techniques. Additionally, it examines existing benchmarks and evaluation frameworks used to assess test generation methodologies. Finally, it highlights research gaps, emphasizing the need for improved benchmarking and evaluation of test generation tools for JavaScript.

3.1 Automated Test Generation

3.1.1 Java

For Java, researchers have made significant advancements in search-based software testing, with its arguably most well-known and proven tool, EvoSuite [20]. EvoSuite, often evaluated with highly optimized evolutionary algorithms like DynaMOSA [48], has undergone extensive trials in various studies and tool competitions, showing its effectiveness in producing high-coverage unit tests [4] [21] [22] [48] [68] [51] [23] [28].

LLM-based test generation has also gained traction for Java, with recent tools such as ChatUniTest [16], AgoneTest [36], and TestSpark [55] showcasing how effectively large language models can generate unit tests. Various studies demonstrate their ability to achieve high-coverage with readable and meaningful tests [16] [68]. However, challenges remain in robustness, coverage, compilation rate and hallucinations [29] [4].

Researchers have also applied symbolic execution tools to Java for automated test generation, aiming to systematically explore program paths by reasoning about symbolic inputs. Tools such as KEX [3] and UnitTestBot [28] use symbolic execution to generate input values that cover different execution paths, often achieving high branch coverage.

3.1.2 Python

Similarly to Java, Python has recently seen substantial research in automated test generation, particularly in search-based testing, LLM-based approaches and hybrid ones. Pyn-

guin stands out as a dedicated SBST tool for Python, addressing the challenges posed by Python's dynamic typing [37]. Its effectiveness has been shown in winning the SBST unit test competition 2024 [19] and in various evaluation studies [33] [29] [76] [38].

LLM-based test generation has also gained traction, with many recent tools and studies. Tools such as TestForge [29], an LLM-Agent approach to test generation, TELPA [76], an LLM-based tool with program-analysis-enhanced prompting and CoverUp [53], a coverage-guided LLM-based approach, have all shown promising results. Many of these tools compare their performance against Pynguin or the widely studied hybrid approach CODAMOSA [33]. CODAMOSA, tested across various benchmarks, has demonstrated how pre-trained language models can enhance search-based test generation by overcoming coverage plateaus.

3.1.3 JavaScript

Despite JavaScript's widespread use [2] [1], automated unit test generation for these languages has not seen a single tool gain significant traction. While various tools have been researched, none have achieved the level of adoption or evaluation seen for the above mentioned Java and Python tools. A survey on JavaScript test generation highlights the unique challenges posed by its dynamic typing, asynchronous execution, prototype-based inheritance, and event-driven nature, making it a complex target for automated testing [7]. These challenges, as well as the distinction between client-side (browser-based) and server-side (Node.js-based) JavaScript, have led to many specialized tools instead of ones that generalize across different environments.

Artemis

Artemis is one of the earliest automated test generation tools for JavaScript web applications [11]. It uses feedback-directed random testing, generating test inputs based on execution data from previously generated inputs. However, it is focused on event handlers in web applications and does not support modern JavaScript features like async/await or modules, limiting its applicability to modern projects.

SymJS

SymJS is a test generation tool that applies symbolic execution to JavaScript, allowing it to systematically explore different execution paths [34]. However, like Artemis, SymJS was designed with a focus on the generation of event sequences in client-side JavaScript, limiting its applicability to modern Node.js applications.

JSEFT

JSEFT was also developed for event-driven testing for web applications, but it additionally supports function level test generation [41]. By analyzing the elements of the web application, JSEFT extracts the states of JavaScript functions to create function-level unit tests.

LambdaTester

LambdaTester was created to handle higher-order functions, which are common in JavaScript [58]. As such, it is particularly useful for testing functions that accept other functions as arguments or return functions. LambdaTester does this by generating and improving callback functions as inputs to the methods under test, focusing on how the callback is invoked.

Nessie

Nessie is also a unit test generation tool designed to support higher order functions [10]. However, unlike LambdaTester that only addresses the sequencing of method calls, Nessie offers an approach for nesting API calls by using a tree structure instead of a simple sequence to represent test cases. Additionally, while LambdaTester only handles synchronous callbacks, Nessie covers both synchronous and asynchronous callbacks. This is made possible through an automated API discovery phase that identifies which API functions accept either type of callback.

SynTest

SynTest is a recent search-based test generation tool that utilizes unsupervised probabilistic type inference approach to infer data types during the test generation process [67]. The inference is done using one of two strategies, proportional sampling or ranking, both of which rely on type hints obtained through static analysis. These inferred types are then integrated into the main loop of a search algorithm like DynaMOSA, a state-of-the-art genetic algorithm for many-objective optimization [48].

TestPilot

TestPilot is another recent tool that applies LLM-based test generation for JavaScript unit testing [56]. Unlike traditional methods that require additional training or few-shot learning on example tests, TestPilot generates tests without the need for extra training or manual effort. The tool takes a function's signature and implementation, along with usage examples extracted from documentation, as input prompts for the LLM. If a generated test fails, TestPilot re-prompts the model with the failing test and error message to produce a new test that addresses the issue.

Summary

In summary, earlier JavaScript testing tools often targeted narrow use cases or older language versions, limiting their relevance for general-purpose unit testing. Recent tools aim for broader applicability, but they have not been thoroughly evaluated or benchmarked against each other. This thesis aims to address this gap by systematically benchmarking modern JavaScript test generation tools.

3.2 Current Benchmarks and Evaluation

3.2.1 Benchmarks

Benchmarks for automated test generation differ significantly in their curation methods, focus, and adoption across research. While some have become widely used across multiple studies, others are tied to specific tools. The selection process varies, with some prioritizing real-world diversity and complexity, others focus on fault detection with reproducible bugs and many are constrained by tool-specific requirements.

SF110

One of the most widely used benchmarks, SF110, is a collection of Java projects from SourceForge with a methodology designed to reduce bias [21]. It included both randomly selected projects and the 10 most popular Java projects to counterbalance the risk of including inactive, poorly maintained, or low-quality code. The study demonstrated how benchmark selection directly impacts evaluation results, as using arbitrary projects could skew findings in favor of certain tools. To ensure reliability and variance, SF110 analyzed projects with static analysis, measuring lines of code, number of files, testable classes, branches, and cyclomatic complexity to provide a detailed characterization of the dataset.

SF110 has been widely used in tool evaluations [21] [47] [48] [68] [25] [62] [46]. However, this widespread use raises concerns about potential bias, as tools like EvoSuite [20] may have been fine-tuned on the dataset, potentially compromising the validity of evaluation results [4]. Additionally, studies have raised concerns on its true representativeness as it mainly consists of low-complexity classes and small projects [71].

Pynguin's Benchmark

Other benchmarks are made with specific research questions in mind. Pynguin's Python benchmark prioritized projects with type annotations, specifically to investigate how type information influences test generation [38]. Similarly to SF110, it applied static analysis to assess code complexity, specifically measuring lines of code, number of modules, objects, predicates, and types detected per project.

ChatTester's Benchmark

ChatTester's benchmark, another Java-focused benchmark, started with 4,685 GitHub projects but filtered them to 185 well-maintained, high-quality projects based on criteria such as continuous updates, popularity (100+ stars), and successful compilation in Maven [77]. This approach ensured that the projects were representative of actively used software.

Defects4J

Many benchmarks focus on fault detection rather than general test generation [31] [63] [74] [27]. For example, Defects4J provides real-world Java bugs for mutation-based and fault-detection testing [31]. However, its frequent use in LLM-based studies raises concerns

about data leakage, as many AI models are trained on datasets containing the same projects [4].

Data leakage

Studies like ChatUniTest address the issue of data leakage effects by incorporating newer, unseen projects created after LLM training data was finalized, allowing for a more reliable assessment of a model's generalization ability [16]. Data leakage concerns is also one of the motivating factors of the creation of the GitBug-Java dataset, made up of only recent real-world bugs [63]. As such, it has been used by recent comparative studies to mitigate the effects of data leakage on evaluation results [4].

3.2.2 JavaScript benchmarks

Unlike Java and Python, JavaScript lacks a standardized benchmark for test generation evaluation, and generally has far fewer benchmarks [71]. Instead, existing benchmarks are tailored to specific tools and evaluation needs.

Tool-Specific Benchmarks Some benchmarks are tailored to particular use cases. For example, the benchmarks used in *Artemis* and *JSEFT* are centered on web-based applications, whereas *LambdaTester* and *Nessie* focus on higher-order functions. Although these benchmarks are useful for their intended domains, they do not generalize well to broader categories of software projects.

SynTest Benchmark SynTest incorporates static analysis to ensure selected projects meet a minimum cyclomatic complexity threshold of > 1, thereby avoiding trivial units [49] [48]. Although GitHub popularity metrics such as stars and forks are used to guide project selection, it includes only a few projects, which appear cherry-picked to align with the strengths of the tool itself. This raises concerns about potential bias, as the benchmark may be partially engineered for the tool rather than serving as a general-purpose evaluation suite.

TestPilot Benchmark By also mining GitHub for projects, the TestPilot benchmark raises concerns about data leakage, as many models are known to have trained on GitHub repositories [54]. However, it mitigates this by including and evaluating projects from GitLab in a separate analysis. Despite these efforts, TestPilot does not account for the complexity of units under test. This limits the ability to draw conclusions about the tool performance across varied levels of code complexity.

ProjectTest Benchmark The ProjectTest benchmark shares similar shortcomings [73]. It consists primarily of small projects selected based on GitHub stars, without clear strategies to prevent data leakage despite being intended for evaluating LLM-based test generation. It also lacks complexity-based filtering, meaning the benchmark may not accurately reflect the diversity and difficulty of real-world codebases.

Challenges of a Generic JavaScript Benchmark JavaScript spans a wide range of frameworks, runtime environments, and domains, from frontend web applications to backend server-side code and diverse npm libraries. This diversity makes it difficult to design a single benchmark that fairly represents all use cases. For this thesis, we focus specifically on backend JavaScript and npm libraries, allowing us to evaluate test generation tools in a coherent and relevant subset of the language while acknowledging that conclusions may not fully generalize to other domains.

Summary

Existing benchmarks highlight best practices and pitfalls in evaluating automated test generation tools. While some argue that diverse real-world projects capture language features, this cannot be verified without systematic static analysis. Ultimately, the lack of a standardized, broadly used benchmark for JavaScript has led to inconsistencies in evaluations, hindering fair comparison and progress in this field.

3.2.3 Existing comparative studies

Several studies have evaluated both search-based software testing (SBST) and LLM-based test generation, though mostly in the context of languages like Java and Python. Some focus on comparing SBST techniques against each other [49] [38], while others assess the effectiveness of different LLM-based approaches [72] [36] [73]. Additionally, tool competitions have been used to compare these methods, with some for Java and Python [51] [23] [28] [19], but none have been conducted for JavaScript.

Fairly comparing SBST, LLM-based, and hybrid approaches in test generation presents a significant challenge, as these methods rely on fundamentally different underlying methodology and algorithms. SBST techniques typically explore the input space through guided search, while LLM-based approaches leverage pretrained language models to generate code based on natural language prompts or context. Hybrid methods may combine elements of both, further complicating evaluation. Despite these differences, a few studies have proposed methodologies that enable meaningful and fair comparisons across these paradigms, setting a standard for research in this direction [4] [62] [26] [29] [68].

Despite these efforts, no comprehensive comparative study exists for JavaScript. While research has examined SBST and LLM-based tools individually, there is no established framework for evaluating them side by side in these languages.

3.3 Research Gap

To the best of our knowledge, there is no established, widely used benchmark for JavaScript. Existing benchmarks focus on specialized subsets of JavaScript applications, or are curated with lack of thought for diversity, complexity and often without considering comprehensive language representation. This lack of standardization makes it difficult to fairly evaluate generalized test generation approaches, particularly those that need to handle JavaScript's diverse and challenging features [7].

Furthermore, while comparative studies have assessed SBST and LLM-based test generation for other languages, no proper evaluations exist for JavaScript. As new tools emerge, their effectiveness remains unverified in a structured, comparative manner, leaving a gap in understanding how different approaches compare in these languages.

Therefore, in this study, we propose a benchmark and comparative evaluation framework tailored for JavaScript unit test generation. Our approach focuses on assembling a diverse and representative dataset of real-world projects, incorporating static analysis to ensure variation in complexity and language features. By comparing both SBST and LLM-based tools on this curated benchmark, we aim to provide the first structured comparison of their effectiveness, strengths and weaknesses in these languages. This work lays the foundation for more consistent evaluation practices and contributes to furthering automated test generation research for JavaScript.

Chapter 4

Benchmark Design

This chapter introduces the first part of our methodology, the design and curation of a benchmark tailored for evaluating JavaScript test generation tools.

Our goal is to curate a dataset that is diverse, complex, representative of real-world coding practices, and mindful of language-specific features and evaluation bias. This chapter describes the process in four stages: the initial collection of candidate projects, the identification of JavaScript features of interest, the static analysis used to assess feature coverage and complexity, and the final filtering that produces a balanced benchmark capable of supporting fair and meaningful comparisons between search-based and LLM-based test generation tools. Figure 4.1 summarizes the workflow of the benchmark design described above.

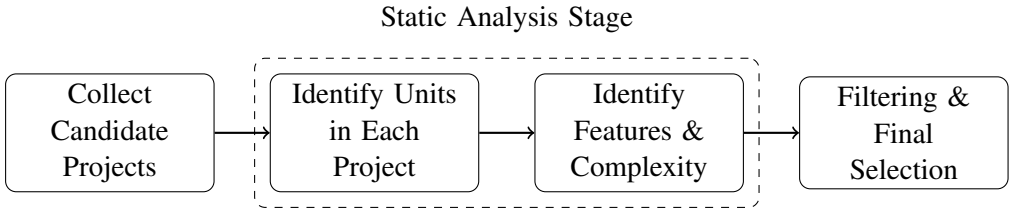


Figure 4.1: Pipeline illustrating the curation steps of the JavaScript benchmark.

4.1 Project Collection Strategy

To build the benchmark, we collected both JavaScript and TypeScript candidate projects from a variety of sources. The initial pool included projects from existing benchmarks [56] [67] [13], as well as the most popular and actively maintained open-source projects hosted on GitHub and GitLab. Using GitLab as an additional source helps mitigate data leakage, since many models are trained on GitHub repositories [54].

We based the selection of projects from GitHub and GitLab on the number of stars, which serves as an indicator of project quality [14]. Additionally, all projects were required

to successfully install and build using standard package managers, such as npm or pnpm, to ensure environment compatibility for test execution.

The curated dataset includes both JavaScript and TypeScript projects. Since TypeScript is largely a superset of JavaScript, we treat TypeScript code as JavaScript for the purposes of analysis, noting tool limitations only where parsing TypeScript-specific constructs causes failures.

4.2 Features representing JavaScript

To design a benchmark that effectively evaluates test generation tools, we first identify key JavaScript features which can influence testing difficulty. JavaScript's flexible nature introduces unique characteristics that distinguish it from statically-typed, class-based languages like Java. These features, such as dynamic typing, asynchronous behavior, and prototype-based inheritance, pose challenges for automated test generation and can significantly impact both the complexity of units and the coverage of test cases.

We selected the features discussed here based on the JavaScript documentation [1] and a survey on dynamic analysis and test generation for JavaScript [7]. These sources help highlight the most challenging aspects of the language that can influence test generation. The features either introduce non-trivial execution paths or create difficult-to-assert properties within test cases, making them essential for evaluation in a benchmark.

Based on these resources, we consider the following features for our benchmark collection:

- **Dynamically Typed:** JavaScript's dynamic typing makes it challenging to infer types, especially when dealing with function parameters or object properties. This ambiguity complicates property access and control flow, making it harder to explore execution paths effectively.

```
function add(a, b) {
    return a + b; // Could be number addition or string
        concatenation
}
console.log(add(2, 3));    // 5
console.log(add("2", 3)); // "23"
```

- **Prototype-Based Inheritance:** Unlike class-based languages, JavaScript relies on prototype chains, where objects can inherit properties and methods from other objects. Tools need to handle dynamic property lookups and inheritance chains effectively to ensure full coverage in test cases.

```
function Animal(name) { this.name = name; }
Animal.prototype.speak = function() {
    console.log(this.name + " makes a noise.");
};
```

[1] https://developer.mozilla.org/en-US/docs/Web/JavaScript

```
function Dog(name) {
    Animal.call(this, name);
}
Dog.prototype = Object.create(Animal.prototype);

new Dog("Rex").speak();
```

- **Asynchronous Programming:** JavaScript's asynchronous nature, with constructs like Promises and async/await, introduces additional complexity. Test generation tools must consider both asynchronous error handling (e.g., `.catch()`) and synchronous error handling (e.g., try-catch), as well as the potential for race conditions or non-deterministic behavior that could affect execution paths.

```
async function fetchData() {
    try {
        let res = await fetch('https://example.com');
        console.log(await res.text());
    } catch (err) {
        console.error("Error:", err);
    }
}
fetchData();
```

- **Closures:** JavaScript closures allow variables to persist within the scope of a function, even after it has finished executing. This behavior makes it difficult to test the interactions between functions and their captured variables, particularly when closures are nested or passed as arguments.

```
function makeCounter() {
    let count = 0; // 'count' lives in the closure's scope
    return function() {
        count++; // Still accessible even after makeCounter()
            returns
        return count;
    };
}

const counter = makeCounter();
console.log(counter()); // 1  (count starts at 0, then increments)
console.log(counter()); // 2  (count is preserved between calls)
```

- **Higher-Order Functions:** JavaScript's support for higher-order functions that can accept other functions as arguments or return functions, introduces new layers of complexity. This feature can lead to highly nested, dynamic execution paths that are challenging to capture and test.

```
function applyOperation(a, b, operation) {
    return operation(a, b);
}
console.log(applyOperation(3, 4, (x, y) => x * y));
```

- **Dynamic Object and Property Manipulation:** JavaScript allows the modification of objects at runtime, including the addition and removal of properties. This dynamic manipulation can create unpredictable behaviors, complicating test case generation, especially when properties are added or modified based on runtime conditions.

```
const obj = {};
obj.newProp = 42;
delete obj.newProp;
```

- **Permissive and Weak Typing Semantics:** JavaScript's permissive runtime behavior and weak typing introduce implicit type coercion, which can lead to unexpected behaviors.

```
console.log('1' + 1); // "11"
console.log('5' - 2); // 3
```

- **Non-Fixed Function Parameters:** JavaScript allows functions to accept a variable number of parameters, making it difficult to generate test cases that cover all possible parameter combinations.

```
function logAll(...args) {
    console.log(args);
}
logAll(1, 2, 3);
```

- **Global Variables via Implicit Declaration:** JavaScript allows for implicit global variable declarations, which can lead to hidden dependencies and unpredictable behaviors in test cases.

```
function setGlobal() {
    globalVar = 10; // No var/let/const => becomes global
}
setGlobal();
console.log(globalVar);
```

- **Undefined Handling:** In JavaScript, accessing a non-existent property on an object doesn't throw an error but instead returns undefined.

```
const user = {};
console.log(user.name); // undefined
```

- **Version Variations:** JavaScript evolves rapidly, and different versions of the language introduce different syntax and behavior.

```
// ES5
var fs = require('fs');

// ES6+
import fs from 'fs';
const square = n => n * n;
```

- **Browser Integration:** Since JavaScript is commonly used for web development, many projects interact with the DOM.

```
document.getElementById('btn').addEventListener('click', () => {
    alert('Button clicked!');
});
```

4.3 Static Analysis and Feature Extraction

To guide the design and curation of our benchmark, we use static analysis to both identify the units under test and extract metrics on their JavaScript language features and complexity. These metrics quantify the diversity and challenge of each project, supporting the selection of a balanced and representative benchmark.

4.3.1 Tooling

We developed a set of custom, reusable ESLint plugins to perform this static analysis. ESLint is a widely used pluggable linter for JavaScript and TypeScript that parses source code into an abstract syntax tree (AST), allowing rules to traverse and analyze code patterns. Our plugins use this infrastructure to detect units, specific language features and complexity metrics in a lightweight and extensible way [2]. These plugins:

- Identify and extract units under test
- Detect the presence of targeted JavaScript features within those units
- Compute metrics such as LOC and cyclomatic complexity.

We implemented a total of 18 plugins, each responsible for detecting a unit, feature or computing a specific metric. While the plugins successfully detect all features listed above, some language constructs in JavaScript are inherently dynamic and cannot always be fully captured through static analysis (e.g., closures and global variables). Nevertheless, our approach ensures that all statically identifiable features relevant for evaluating test generation tools are correctly extracted.

The plugins are modular and designed for extension. They form a key contribution of this work and can be reused or enhanced by future researchers and practitioners. Our static analysis plugins are publicly available on GitHub[3].

4.3.2 Unit Identification

Our static analysis tool automatically detects units under test across projects. However, unlike Java, where units under test are always classes, JavaScript allows for more flexible structures. In this study, we define a unit under test as any exported function, method, or

[2] https://eslint.org/

[3] https://github.com/SagaRut/dataset-evaluation

class, since exportability ensures the unit can be directly tested. This approach aligns with JavaScript's functional nature and ensures compatibility with both test generation tools and the static analysis pipeline.

4.3.3 Feature Identification

We focus on detecting the JavaScript language features outlined in the section above. For example, to identify the use of asynchronous programming, we look for reserved keywords such as `async` and `await` within function declarations and expressions. Similar keyword- and pattern-based heuristics are applied for detecting other features, ensuring consistent and lightweight static analysis across projects.

For each project, we identify:

- **Which features are present** in the units under test,
- **How frequently** each feature appears (e.g., number of units using async/await and how frequently within a unit a feature appears),
- **How features co-occur** within the same unit or across a project.

This allows us to not only confirm the presence of each feature but also assess the variety of combinations i.e. feature interactions that are important for testing the robustness of test generation tools.

4.3.4 Complexity Metrics

In addition to language features, we evaluate units based on standard software complexity metrics:

- **Cyclomatic Complexity [40]:** Captures the number of linearly independent paths through a unit, reflecting branching logic and test case requirements.
- **Lines of Code (LOC):** Provides a rough measure of implementation size.
- **Number of Testable Units:** Reflects the overall size of the project's exposed API.

Projects with higher average complexity and greater variability in these metrics present more significant challenges for test generation tools [75]. However, to ensure realism, we avoid artificially complex or synthetically generated code, all projects are real-world, open-source libraries.

4.3.5 Diversity and Complexity Scoring

The analysis results are further processed to compute diversity and complexity scores for each unit, project and for datasets as a whole. To measure this, we generate a feature matrix, where each row corresponds to a unit under test and each column to a specific language feature. The matrix is binary-valued, indicating whether a unit exhibits a given feature.

From this, we compute:

- **Feature coverage:** The proportion of targeted features represented in the dataset.
- **Feature entropy:** A measure of how evenly features are distributed across units, indicating balanced representation.
- **Feature density:** The proportion of all possible feature–unit combinations where a feature is present, capturing how densely features appear throughout the dataset.

While a higher average number of features per unit can indicate greater diversity, this alone is not a sufficient benchmark goal. Our primary focus is on achieving high overall coverage across features and maximizing the diversity of feature combinations. Datasets that include all target features, even if each appears only a few times, are more useful than datasets that overrepresent a limited subset.

Using the feature matrix and the previously computed complexity metrics, we define scoring mechanisms at three levels:

- **Unit level:** Each unit is scored based on the number of features it contains, its cyclomatic complexity, and lines of code.
- **Project level:** A project's score aggregates the diversity and complexity of its units, reflecting how challenging and representative the overall project is.
- **Benchmark level:** At the benchmark level, we evaluate the overall feature coverage, distribution of complexity, and diversity of feature combinations across all included projects.

4.4 Filtering and Final Selection

The purpose of this stage is to select a final set of projects that form a balanced and representative benchmark. From the initial pool of 300 projects, we evaluated each project based on feature diversity and unit complexity. The final benchmark was curated using specific thresholds for both criteria. While the framework allows for user-defined thresholds, a well-rounded benchmark should ideally cover all targeted language features and span a broad range of complexity levels.

For our curated benchmark, we define feature completeness at both the project and benchmark levels, ensuring that all selected projects collectively span the full set of JavaScript features. To promote feature interactions, we also enforce a minimum threshold on the average number of language features per unit. Additionally, we set minimum thresholds for cyclomatic complexity to ensure the selected code poses meaningful challenges for test generation tools.

A more detailed breakdown of the final benchmark composition and statistics such as feature distribution, complexity ranges, and language proportions is presented in the evaluation section 5. In the results section 6 we also compare our benchmark against existing ones to highlight differences in complexity and diversity.

Custom Filtering and Subsets

The collection of analyzed projects can also be tailored to fit specific evaluation goals. For example, one might filter projects that use higher-order functions to evaluate tools like LambdaTester [58] and Nessie [10] made specifically to handle such, or select projects involving DOM manipulation to assess browser-based testing capabilities. Subsets can be created by specifying desired features, complexity thresholds (e.g., projects where all units have complexity above a given value), or a particular programming language (JavaScript vs. TypeScript). Filtering can also be applied at the unit level. If a project contains units that do not meet certain criteria, one may choose to exclude them.

Chapter 5

Empirical Evaluation

This chapter provides an overview of the evaluation setup used to compare automated test generation tools. It outlines the benchmark used, tool configurations, and the shared pipeline used to measure coverage and feature-based performance. The setup is designed to directly address our research questions and ensure a fair, reproducible comparison across tools.

5.1 Benchmark

For the empirical evaluation we construct and use a curated benchmark designed to represent the diversity, complexity, and the practical relevance of real-world codebases. This benchmark is the result of the methodology described in the previous chapter 4, focusing on maximizing feature completeness and complexity.

The final benchmark consists of 42 open-source projects sourced from both GitHub and GitLab. These projects were selected from the initial pool of evaluated projects based on a well-defined set of criteria, ensuring the benchmark remains tool-agnostic and unbiased. The benchmark is feature-complete, containing representative examples of all JavaScript features outlined in Section 4.2. It also maintains a high level of complexity, with each project exhibiting an average cyclomatic complexity greater than 2 per unit and an average of more than 3 JavaScript features per unit for interaction richness. The cyclomatic complexity aligns with established guidelines [49] [48], which recommend that units used for evaluating test case generation tools should be non-trivial, i.e. contain multiple execution paths. To guarantee the validity of the benchmark, each project's dependency and build environment was verified by running npm/pnmp/yarn install and building the project where applicable.

Benchmark Statistics

The curated benchmark includes:

- **42 projects**: Selected from GitHub (40) and GitLab (2)
- **2850 units under test**: Defined as exported functions, methods, or classes.

- **Feature complete**: Every JavaScript feature defined in the previous chapter is represented at least once
- **Average cyclomatic complexity of units:** > 2
- **Average JavaScript features per unit:** > 3

With Table 5.1 additionally depicting the feature density, average number of features per project, feature entropy, average cyclomatic complexity and average number of features per unit.

Table 5.1: Feature Coverage Metrics of the Custom Benchmark

Metric	Benchmark
Number of Projects	42
Features Covered	15
Feature Density	72.1%
Avg. Features / Project	10.81
Feature Entropy	4.32
Average Cyclomatic Complexity	3.33
Total Units	2850
Avg. Features / Unit	4.32

Project Summary

Table 5.2 is a summary of all projects included in the benchmark including relevant complexity and diversity metrics. Namely, the number of units, the average cyclomatic complexity of units and the average number of JavaScript features present per unit. Each project entry also includes a direct link to its repository along with the specific commit hash used for the benchmark to ensure reproducibility and traceability.

Figure 5.1 shows the distribution of JavaScript features across the benchmark projects. Each bar represents a distinct feature, and the y-axis indicates how many projects in the benchmark include that feature.

5.2 Test Generation Tools

This study compares the test generation capabilities of two state-of-the-art tools representing distinct approaches: search-based software testing and large language model-based generation. We chose these tools for this comparison, over the others mentioned in Chapter 3, as they are the two newest tools of the two most researched approaches (SBST and LLM-based), while also being the current state-of-the-art, reportedly performing well over diverse JavaScript code. The two tools are SynTest [67], a modern SBST tool for JavaScript, and TestPilot [56], a recent LLM-based tool with iterative refinement designed specifically for JavaScript and TypeScript unit test generation. While both tools target unit test creation for

Table 5.2: Summary of Projects in the Benchmark

Project	**# Units**	**Avg. CC**	**Avg. No. Features**
AdminLTE	11	2.33	3.91
autoprefixer	83	2.82	4.11
bower	96	2.17	4.34
dropzone	5	2.73	6.40
express-jwt	3	2.00	5.67
extract-text-webpack-plugin	13	2.91	3.08
hoodie	20	2.85	3.90
json-server	4	2.64	3.75
knex	112	2.02	5.01
lodash	7	3.71	8.14
markdown-here	8	3.75	3.63
mdx	38	3.26	3.79
nedb	86	4.41	3.52
release-it	30	2.27	4.00
samples	4	2.00	5.25
ScrollMagic	6	5.00	4.33
scrollreveal	27	4.52	3.41
shelljs	48	4.92	4.83
sitespeed.io	165	4.82	3.59
spectacle-code-slide	3	3.00	5.00
supertest	5	2.93	3.20
tabulator	147	2.10	3.63
postcss	27	3.65	5.44
puppeteer	329	2.27	3.48
terminus	383	2.03	3.96
vue-devtools	226	2.00	3.04
incubator-weex-ui	7	2.43	4.00
ZY-Player	28	2.40	4.11
commander.js	4	3.83	6.00
express	12	3.22	3.17
javascript-algorithms	29	2.17	4.21
crawler-url-parser	3	12.0	6.67
delta	8	5.87	6.25
node-dir	6	3.33	3.00
node-glob	16	3.16	3.44
node-graceful-fs	3	5.67	7.00
spacl-core	4	2.38	5.00
zip-a-folder	2	3.67	3.00
Beatbump	176	2.49	3.34
felte	174	3.05	3.29
pandora	184	2.18	3.33
xyflow	308	3.01	3.04

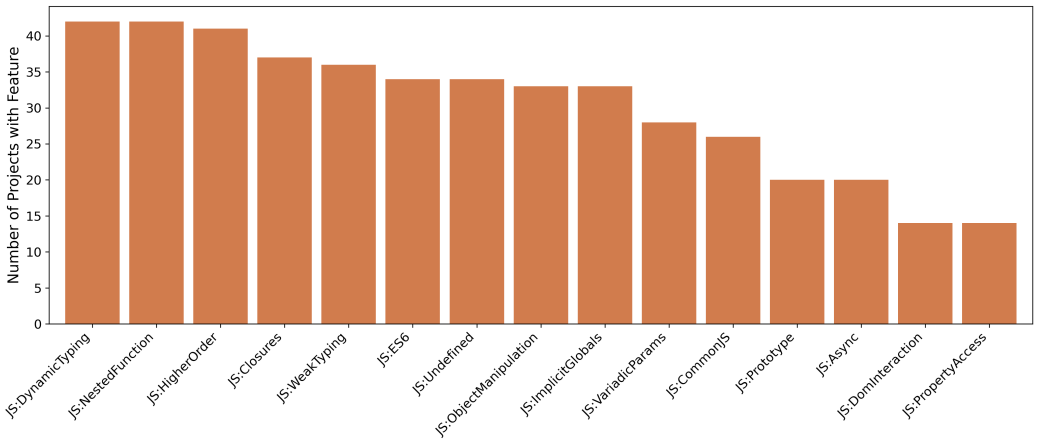


Figure 5.1: Distribution of JavaScript features across benchmark projects.

Node.js projects using the Node Package Manager [1] and generate tests in the Mocha testing framework [2], they differ in how they assess coverage. TestPilot uses Istanbul [3], whereas SynTest relies on a modified custom version of Istanbul with additional instrumentation.

In addition to these two tools, we also compare initial performance results to a zero-shot version of TestPilot, representing a naive LLM approach without execution feedback or prompt refinement. This baseline helps contextualize the gains achieved by TestPilot's more advanced iterative techniques.

These two tools represent the two newest state-of-the-art in their respective approach. However, despite their renown, these tools have not yet been evaluated against each other or against a common benchmark. Our comparison aims to fill this gap by examining their performance across a diverse and complex set of real-world JavaScript projects.

5.3 Tool Setup

To address the inherent nondeterminism of the automated test generation approaches under evaluation, we executed each tool 10 times per project, as suggested by existing guidelines [8]. The time budget of SBST tools can significantly effect the quality of tests generated [50]. Thus, we imposed a time budget of 120 seconds on SynTest per execution, aligning with prior studies [4]. While setting a time constraint like this for evaluating SBST is common, it is not natively supported by LLM-based tools. However, in practice, we observed that TestPilot completes generation within 1–2 minutes per UUT, comparable to the time budget imposed on SynTest. As illustrated in Figure 5.2, timestamp screenshots from a representative run confirm this behavior. This execution pattern was consistent across all runs. This form of time budget mitigation for LLM-based tool when comparing to SBST tools has been applied in other recent studies [4].

[1] https://www.npmjs.com
[2] https://mochajs.org/
[3] https://istanbul.js.org/

```
[27.06.2025 06:55.21.334] [LOG]   test_33.js (for quill-delta.AttributeMap.compose at temperature 0, 7 snippets available): PASSED
[27.06.2025 06:55.34.146] [LOG]   test_34.js (for quill-delta.AttributeMap.compose at temperature 0, 7 snippets available): PASSED
[27.06.2025 06:55.54.618] [LOG]   test_35.js (for quill-delta.AttributeMap.compose at temperature 0, 7 snippets available): PASSED
[27.06.2025 06:56.12.531] [LOG]   test_36.js (for quill-delta.AttributeMap.compose at temperature 0, 7 snippets available): PASSED
[27.06.2025 06:56.30.185] [LOG]   test_37.js (for quill-delta.AttributeMap.diff at temperature 0, 2 snippets available): FAILED
```

Figure 5.2: Timestamp screenshots from a representative run of TestPilot showing that generation completes within 1–2 minutes per Unit Under Test (UUT)

Each tool in our experiments comes with numerous configurable parameters that can influence test generation outcomes. To ensure a fair and practical comparison, we use each tool's recommended default settings, as these are widely adopted in the literature and provide reasonable performance without the added cost of extensive parameter tuning. [9]

5.3.1 SynTest Configuration

We ran SynTest with its default configuration, which corresponds to the recommended DynaMOSA preset, a state-of-the-art genetic algorithm designed for efficient multi-objective optimization [48]. The setup included a population size of 50 with type inference enabled in proportional mode to guide the generation process.

5.3.2 TestPilot Configuration

The original TestPilot paper demonstrated the best results using GPT-3.5 Turbo. To stay true to the tool's intended cheap-but-good performance, we chose to evaluate TestPilot with GPT-4o Mini as OpenAI recommends it as the direct replacement for GPT-3.5 Turbo [45]. GPT-4o Mini provides comparable latency and cost-effectiveness but improved output quality. This choice ensures our experiments remain faithful to the original tool's design while using the most current suitable model. Due to model differences however, the system prompt had to be adapted to enforce the expected output format. The final configuration used a static system prompt:

```
// Finish the test and only respond in code, no text and no ```javascript
    ```
```

The user prompt is comprised of the beginning of a unit test, in this case a partial Mocha test including the function under test signature, example below, augmented with additional context depending on the prompt refinement strategy. This may involve providing the function implementation, usage examples extracted from documentation, and, in some cases, re-prompting the model with previously generated failing tests and their corresponding error messages.

```
let mocha = require('mocha');
let assert = require('assert');
let autoprefixer = require('autoprefixer');
// autoprefixer.info()
describe('test autoprefixer', function() {
    it('test autoprefixer.info', function(done) {
```
```

We validated the performance of the tool using the adjusted model and prompt by re-running it on its original benchmark, following the same evaluation setup as in the original paper. This included averaging results over 10 runs to ensure consistency to the original paper results. As shown in Table 5.3, GPT-4o Mini achieves slightly lower performance across all reported metrics, test pass rate, statement coverage, and branch coverage, compared to GPT-3.5-Turbo. However, due to the lack of full per-run data for the original model, more fine-grained comparisons are limited.

The aforementioned zero-shot version of TestPilot was configured by disabling its prompt refinement techniques, resulting in a simple non-iterative method that relies solely on minimal context information, specifically method signatures.

	gpt-3.5-turbo				*gpt-4o-mini*			
Project	**Tests**	**Passing**	**Stmt Cov**	**Branch Cov**	**Tests**	**Passing**	**Stmt Cov**	**Branch Cov**
bluebird	370	204 (55.2%)	68.0%	50.0%	352	136 (38.6%)	58.9%	41.2%
complex.js	209	121 (58.0%)	70.2%	46.5%	252	137 (54.3%)	64.1%	42.1%
countries-and-timezones	28	13 (46.4%)	93.1%	69.1%	32	15 (46.4%)	93.4%	68.6%
crawler-url-parser	14	2 (14.3%)	51.4%	35.0%	17	1 (5.8%)	42.5%	29.4%
quill-delta	152	33 (21.7%)	73.0%	64.3%	190	31 (16.3%)	70.4%	63.5%
gitlab-js	141	14 (9.9%)	51.7%	16.5%	139	10 (7.2%)	65.8%	24.8%
image-downloader	5	4 (80.0%)	63.6%	50.0%	5	0 (0.0%)	20.2%	16.7%
js-sdsl	409	46 (11.3%)	33.9%	24.3%	431	58 (13.4%)	40.1%	41.9%
memfs	1037	471 (45.4%)	81.1%	58.9%	883	460 (52.1%)	65.7%	45.6%
node-dir	40	19 (48.1%)	64.3%	50.8%	40	19 (47.3%)	48.9%	38.1%
dirty	70	32 (45.3%)	74.5%	65.4%	86	41 (47.4%)	61.1%	49.3%
fs-extra	471	277 (58.8%)	58.8%	38.9%	429	210 (49.0%)	51.3%	34.7%
geo-point	76	50 (65.8%)	87.8%	70.6%	66	30 (45.1%)	13.4%	39.0%
glob	68	18 (26.5%)	71.3%	66.3%	58	0 (0.0%)	0.0%	0.0%
graceful-fs	345	177 (51.4%)	49.3%	33.3%	355	176 (49.6%)	30.5%	21.9%
jsonfile	13	6 (48.0%)	38.3%	29.4%	14	0 (0.0%)	0.0%	0.0%
uneval	7	2 (28.6%)	68.8%	58.3%	6	2 (34.3%)	62.5%	55.6%
omnitool	1033	330 (31.9%)	74.2%	55.2%	815	288 (35.3%)	68.5%	53.4%
plural	13	8 (61.5%)	73.8%	59.1%	15	8 (52.2%)	75.1%	66.7%
pull-stream	83	34 (41.0%)	69.1%	52.8%	88	32 (36.2%)	72.2%	54.9%
q	323	186 (57.6%)	70.4%	53.7%	382	187 (48.9%)	69.9%	53.8%
rsvp	109	70 (64.2%)	70.1%	55.3%	134	91 (67.7%)	74.5%	61.2%
simple-statistics	353	250 (70.9%)	87.8%	71.3%	384	219 (57.0%)	85.6%	73.9%
spacl-core	85	13 (15.3%)	78.3%	50.0%	94	22 (23.5%)	76.2%	47.5%
zip-a-folder	11	6 (54.5%)	84.0%	50.0%	16	0 (0.0%)	22.2%	15.9%
Median		**48.0%**	**70.2%**	**52.8%**		**38.6%**	**62.5%**	**42.1%**

Table 5.3: Comparison of test generation results between GPT-3.5-Turbo and GPT-4o-Mini across benchmark projects.

5.4 Evaluation Pipeline

To ensure consistency and reproducibility across test runs, we designed and implemented an automated evaluation pipeline. Each tool is executed independently within isolated Docker containers. This setup mitigates environment-related variations and simplifies dependency management.

After test generation, the resulting test suites were extracted and executed using the Mocha testing framework [4]. We used Mocha's default timeout of 2 seconds per test case

[4]https://mochajs.org/

to reflect typical developer settings. To evaluate code coverage, we used Istanbul [5], which measures line, statement, function, and branch coverage across the generated tests.

Following execution, we collect both code coverage statistics and test pass rates to quantitatively assess each tool's performance. All results are logged systematically to facilitate our analysis. The overall workflow of our evaluation pipeline is summarized in Figure 5.3. Additionally, the pipeline is publicly available on GitHub[6].

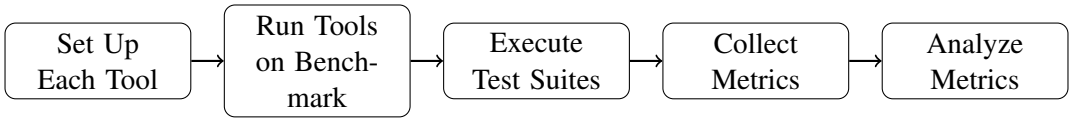


Figure 5.3: Overview of the automated evaluation pipeline.

5.5 Analysis

To evaluate the performance and behavior of the test generation tools, we analyze execution-based metrics, failure types, coverage similarity and do a correlation analysis on code feature-based metrics. To further validate the performance comparison of the three tools, we conducted sound statistical analysis.

5.5.1 Execution-Based Metrics

We assess the quality and usefulness of generated tests through the following execution-based metrics:

- **Test Pass Rate:** The proportion of generated test cases that successfully pass during execution, indicating syntax and assertion correctness.
- **Code Coverage:** We measure code coverage using four standard coverage types:
  - *Line Coverage*: Percentage of executed lines.
  - *Statement Coverage*: Percentage of executed program statements.
  - *Function Coverage*: Percentage of invoked functions.
  - *Branch Coverage*: Percentage of conditional branches exercised.

These metrics allow us to quantify how effectively the tools explore the codebase and generate meaningful, executable tests.

[5] https://istanbul.js.org/

[6] https://github.com/SagaRut/test-generation-evaluation

5.5.2 Statistical Analysis

To assess the significance of observed differences, we conducted pairwise statistical comparisons for each project and metric. For this we used Mann-Whitney U tests [43] for statistical significance combined with Vargha-Delaney $\hat{A}_{12}$ [70] for effect size calculation, in line with prior work [4] [49] [28]. Specifically, for each project–metric pair, we performed three Mann-Whitney U tests : SynTest vs. TestPilot, SynTest vs. TestPilot baseline, and TestPilot vs. TestPilot baseline. Winners in each comparison were determined based on both statistical significance (p-value < 0.05) and a meaningful Vargha-Delaney effect size (small or larger).

5.5.3 Test Failure Types

We assess test outcomes by categorizing failures into the following types:

- **AssertionError:** Indicates that an expected condition within a test case was not met, signaling a test logic failure.
- **FileSystemError:** Represents errors related to file operations, such as missing files or permission issues.
- **ModuleNotFoundError:** Occurs when a required module or dependency cannot be located during test execution.
- **ReferenceError:** Happens when a variable or function is referenced before it has been declared.
- **SyntaxError:** Indicates invalid code syntax preventing test execution.
- **TimeoutError:** Results from tests exceeding the allowed execution time, suggesting potential infinite loops or async issues.
- **TypeError:** Arises when an operation is performed on a value of an inappropriate type.
- **Other:** Captures any failure types not classified above.

This classification allows us to analyze not only the quantity but also the nature of failures generated by each test generation tool, providing deeper insights into their limitations in producing reliable tests.

5.5.4 Similarity Analysis

Furthermore, we analyze how complementary or redundant the tools' coverage is. We assess the overlap in coverage between tools by collecting the sets of statements covered by each test suite and computing the Jaccard similarity (Intersection-over-Union). This is done on three levels: averaged over 10 runs, collectively across all runs, and separately per project. Additionally, we check for coverage containment to determine how much of one tool's coverage is a subset of another's, indicating potential subsumption.

5.5.5 Correlation Analysis

To better understand how code characteristics influence test generation, we perform a correlation analysis between test outcomes and the following code feature metrics:

- **JavaScript Language Features:** We analyze how the presence of various JS/TS features affects tool performance in terms of coverage and pass rate.
- **Cyclomatic Complexity:** Higher complexity can challenge test generation. We examine whether more complex units result in lower coverage.
- **Number of Units:** We examine whether a higher number of units within a project result in lower coverage.
- **Number of Branches:** We additionally examine whether a higher number of branches within a project result in lower coverage.
- **Test Suite Size:** We analyze whether a larger number of generated tests leads to better coverage or simply adds redundancy.

This analysis enables us to identify the specific strengths and limitations of each tool, highlighting the types of JavaScript constructs they handle effectively and the scenarios where they struggle. By examining what each tool excels at and where their test generation capabilities fall short, we gain a clearer understanding of their practical applicability and potential areas for improvement. We use Spearman's rank correlation coefficient for the correlation analysis [57] as our data does not follow a normal distribution according to the Shapiro-Wilk test [61].

To provide a comprehensive understanding of how tool performance correlates with code characteristics, we perform the correlation analysis at both the project and file levels. Project-level analysis captures broader trends across entire codebases, while file-level analysis offers more granular insights into how specific code units and their individual complexity and feature sets impact test generation effectiveness. Furthermore, to improve readability and clarity, we separate the comparisons into two categories: one focusing on complexity metrics and another on feature diversity metrics. This separation allows us to present dedicated heatmaps for each category, highlighting distinct relationships between performance and the different aspects of the code.

It is important to note that correlation analysis cannot be performed on features that are present in all or no projects/files, as there is no variation to analyze.

5.6 Research Questions

The empirical evaluation is guided by the following research questions, each aimed at understanding different aspects of automated test generation for JavaScript. Our evaluation pipeline and experimental design are structured to provide answers to these questions through empirical data collection and analysis.

- **RQ1: To what extent do existing automated test generation tool benchmarks cover unique features of JavaScript relevant to test generation?**

  To address RQ1, we conduct a static analysis of projects within existing benchmarks to identify the presence and frequency of language-specific features, outlined in 4.2, reusing our tool mentioned in 4.3. We then compare these distributions to a custom benchmark curated for this study to highlight representational gaps and ensure a more comprehensive feature coverage.

- **RQ2: How do current JavaScript test generation tools perform on a comprehensive benchmark created with regard to language representation and complexity?**

  To answer RQ2, we apply each test generation tool to our benchmark (5.1) and evaluate their output using the automated evaluation pipeline described in Section 5.4. We analyze test quality and coverage using the methods detailed in section 5.5, including statistical significance tests (Section 5.5.2), failure type classification (Section 5.5.3) and similarity assessments (5.5.4).

- **RQ3: How do tool performance outcomes correlate to code complexity characteristics and JavaScript features?**

  For RQ3, we use the same shared evaluation pipeline to collect quantitative performance metrics. By correlating these results with code features metrics (5.5.5) extracted from each unit under test, we provide a comparison of the relative strengths and weaknesses of LLM-based vs. search-based tools on a fine-grained level.

  Each research question is directly supported by our benchmark design, tool execution framework, and analysis pipeline, allowing for systematic evaluation across a diverse set of code characteristics and test quality indicators.

Chapter 6

Evaluation Results

In this chapter, we present the results of our empirical evaluation to address the research questions outlined in Chapter 5. The goal of this evaluation is to assess and compare automated test generation tools for JavaScript, with a particular focus on their performance across complex and diverse codebases, as well as on identifying correlations between code characteristics and tool effectiveness. The results are organized by research question, with each section reporting the relevant metrics, visualizations and findings.

6.1 Existing Benchmark Evaluation

The first research question, as stated in Chapter 5, is:

> **RQ1:** To what extent do existing automated test generation tool benchmarks cover unique features of JavaScript relevant to test generation?

6.1.1 Benchmark Analysis

Looking purely at JavaScript feature coverage, figures 6.1 and 6.2 reveal gaps in the *SynTest* and *TestPilot* benchmarks. The figures visualize the distribution of the JavaScript features across the benchmarks, where figure 6.1 shows the number of projects containing each feature, while figure 6.2 presents these values normalized to the size of each benchmark. We can see that SynTest is missing integral JavaScript functionality, namely, a representation of its asynchronous ability. TestPilot includes only a small number of projects that demonstrate features such as prototype usage, property access, and DOM interaction. Overall, it shows a much lower proportion of projects containing each individual feature compared to the other benchmarks.

If we look at the SynTest benchmark further by examining table 6.1, we see that the benchmark consists of 5 projects and covers 13 out of the 15 defined JavaScript features. It has the highest average cyclomatic complexity (5.54) and a high feature density (61.3%), with an average of 9.20 features per project and 3.90 features per unit. However, its overall size is relatively small, with only 99 units in total.

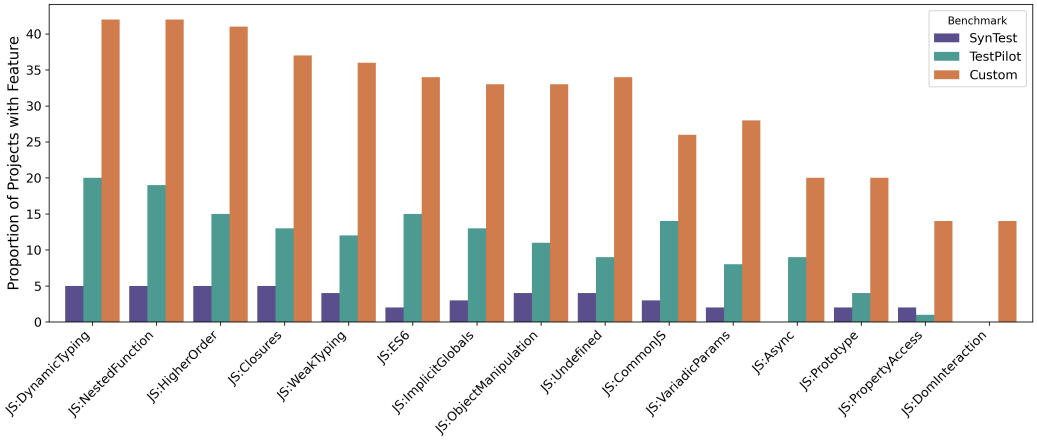


Figure 6.1: JS Feature Coverage Across Benchmarks (Counts)

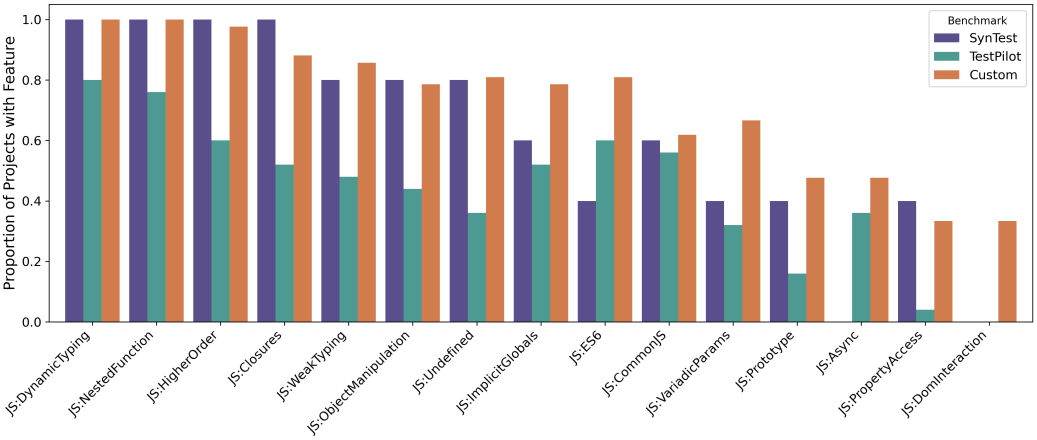


Figure 6.2: JS Feature Coverage Across Benchmarks (Normalized Proportions)

The TestPilot benchmark is significantly larger, comprising 25 projects and 745 units. It covers 14 features but has the lowest feature density (43.5%) and the lowest average features per unit (2.73). While its feature entropy is the highest (6.12), indicating diversity in feature distribution, the average complexity is lower (2.90), and individual projects tend to cover fewer features.

The custom benchmark curated for this study consists of 42 projects and 2,850 units. It includes all 15 target features, with the highest average number of features per project (10.81) and the highest feature density (72.1%), indicating comprehensive and diverse language construct representation. Its cyclomatic complexity (3.33) and feature entropy (4.32) reflect a balance between complexity and feature variety.

Overall, existing benchmarks exhibit trade-offs between size, feature diversity and complexity. SynTest's limited size and absence of asynchronous features pose constraints on thorough evaluation. TestPilot provides scale but at the cost of lower feature density and

complexity. The custom benchmark overcomes these limitations by balancing size, complexity, and comprehensive language feature representation, justifying its use for further evaluations.

Table 6.1: Feature Coverage Metrics Across Benchmarks

Metric	SynTest	TestPilot	Custom
Number of Projects	5	25	42
Features Covered	13	14	15
Feature Density	61.3%	43.5%	72.1%
Avg. Features / Project	9.20	6.52	10.81
Feature Entropy	3.77	6.12	4.32
Average Cyclomatic Complexity	5.54	2.90	3.33
Total Units	99	745	2850
Avg. Features / Unit	3.90	2.73	4.32

Findings: Existing benchmarks vary significantly in their representation of JavaScript features, with some lacking key language constructs essential for thorough evaluation. The custom benchmark developed for this study offers a more comprehensive representation of language features and complexity, providing a stronger foundation for evaluating automated test generation tools.

6.2 Tool Performance Comparison

The second research question, as stated in Chapter 5, is:

> **RQ2:** How do current JavaScript test generation tools perform on a comprehensive benchmark created with regard to language representation and complexity?

6.2.1 Tool Executability

While the custom benchmark consists of 42 projects in total, a significant portion of them could not be executed successfully by the tools. SynTest was only able to run on 12 projects, and TestPilot on 18. Among these, only 10 projects could be run reliably by both tools (Figure 6.3). As a result, this restricted the final benchmark used for comparative evaluation to these 10 shared projects.

The reasons for incompatibility varied between tools. For SynTest, most failures were due to its limited ability to parse certain JavaScript constructs and to handle some TypeScript constructs properly. For TestPilot, the primary issues arose from its API parser expecting a specific export format, namely, the package.json needed to specify a main entry that exported a single default module, unlike other common export conventions. These problems are tool-specific, as thorough validation of each project build and dependency setup was performed, as described in 5.1.

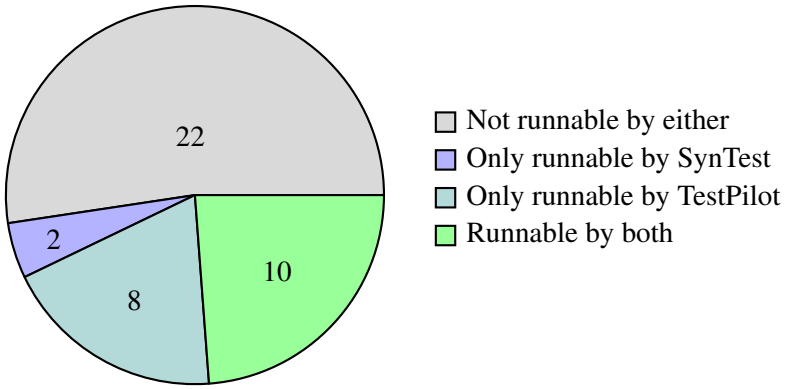


Figure 6.3: Project compatibility across tools. Only 10 out of 42 projects could be run by both tools.

Despite the reduced size, the final benchmark retains much of the feature diversity and structural complexity present in the original set. As shown in Table 6.2, the 10 selected projects collectively cover all 15 targeted JavaScript features. The benchmark maintains a high feature density (69.3%) and exhibits strong average complexity (4.20), ensuring that it still provides a meaningful basis for tool comparison in the following sections.

Table 6.2: Feature Coverage Metrics For Final Benchmark

Metric	**Final Benchmark**
Number of Projects	10
Features Covered	15
Feature Density	69.3%
Avg. Features / Project	10.40
Feature Entropy	3.59
Average Cyclomatic Complexity	4.20
Total Units	252
Avg Number of JS Features per Unit	4.42

6.2.2 Performance Comparison

Table 6.3 presents the per-project performance for each tool in terms of number of test cases and corresponding test pass rates, while Table 6.4 contains the statement coverage and branch coverage obtained on the final benchmark. Comparing the number of tests generated from each tool per project, it is evident that TestPilot utilizes a much more aggressive strategy, generating a huge amount of tests with only a small percentage of them passing. Overall, SynTest achieves the highest median pass rate (66.4%) compared to TestPilot (21.8%) and the TestPilot zero-shot baseline (10%).

However, TestPilot and its zero-shot baseline consistently achieve substantially higher median statement coverage (45.4% and 32.6%) than SynTest (27.9%). Despite the LLM

baseline's somewhat better statement coverage compared to SynTest, its median branch coverage is considerably lower (5.1% vs. 11.2%), indicating that while it covers many statements, its exploration is shallow. In contrast, the full TestPilot tool achieves the highest branch coverage overall (27.9%). In particular, projects like `commander.js` and `supertest` highlight TestPilot's superiority in coverage, while SynTest achieves higher branch coverage primarily on `hoodie`, with a slight advantage on `dropzone` as well.

There is also considerable variation in performance across projects. Branch coverage achieved by SynTest ranges from 1.4% to 31.4%, whereas TestPilot's branch coverage varies more widely, from 0% up to 54.2%. Notably, the baseline LLM tool records 0% branch coverage in five out of the ten benchmark projects, yet reaches as high as 33.9% on others. There is also some similarity in performance in all tools. For instance, all tools perform poorly on `dropzone` while reaching very similar statement coverage on `bower`. Standard deviation values indicate that SynTest shows more variation in pass rates across runs (at most 16.3%), while TestPilot exhibits more variation in coverage metrics on certain projects (at most 16.1%).

Project	**SynTest**		**TestPilot_4omini**		**TestPilot_4omini_baseline**	
	Total # Tests	**Pass Rate (%)**	**Total # Tests**	**Pass Rate (%)**	**Total # Tests**	**Pass Rate (%)**
autoprefixer	48.8 ± 4.7	**47.1 ± 4.9**	9.0 ± 0.8	29.1 ± 5.7	1.80 ± 0.40	40.0 ± 20.0
bower	89.4 ± 2.5	**55.9 ± 1.4**	129.8 ± 4.5	14.4 ± 1.9	35.00 ± 1.00	19.74 ± 3.08
commander.js	148.9 ± 15.7	**99.6 ± 0.5**	567.0 ± 9.5	44.6 ± 0.9	157.30 ± 0.46	47.36 ± 0.84
crawler-url-parser	1.0 ± 0.0	0.0 ± 0.0	18.0 ± 0.0	**3.3 ± 2.7**	2.90 ± 0.30	0.0 ± 0.0
dropzone	11.9 ± 1.1	**94.8 ± 7.9**	220.3 ± 1.6	0.9 ± 0.2	56.80 ± 0.60	0.0 ± 0.0
hoodie	97.2 ± 10.2	**80.1 ± 2.0**	4.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
node-dir	0.0 ± 0.0	0.0 ± 0.0	47.4 ± 1.9	**44.9 ± 6.4**	5.90 ± 0.30	0.0 ± 0.0
node-graceful-fs	9.2 ± 1.5	**76.8 ± 3.6**	533.5 ± 7.4	47.8 ± 0.8	189.30 ± 1.10	52.93 ± 1.63
postcss	243.6 ± 47.9	**83.7 ± 4.4**	327.6 ± 6.2	46.9 ± 1.7	103.50 ± 0.92	58.56 ± 2.22
supertest	3.0 ± 0.0	**46.7 ± 16.3**	386.4 ± 5.1	1.2 ± 0.2	108.20 ± 1.17	0.19 ± 0.37
Median	30.4	66.4	175.0	21.8	45.9	10.0

Table 6.3: Average number of tests and pass rate over 10 runs for SynTest, TestPilot_4omini, and TestPilot_4omini_baseline on selected benchmark projects. Values are reported as mean ± standard deviation.

Project	**SynTest**		**TestPilot_4omini**		**TestPilot_4omini_baseline**	
	Stmt Cov (%)	**Branch Cov (%)**	**Stmt Cov (%)**	**Branch Cov (%)**	**Stmt Cov (%)**	**Branch Cov (%)**
autoprefixer	32.6 ± 5.7	12.9 ± 4.8	**43.7 ± 0.0**	**22.8 ± 0.0**	29.29 ± 14.64	10.28 ± 5.14
bower	30.0 ± 2.3	9.3 ± 1.9	**34.5 ± 1.2**	**15.7 ± 0.7**	32.22 ± 0.23	14.57 ± 0.18
commander.js	14.5 ± 0.6	9.4 ± 0.6	**66.9 ± 3.0**	**54.2 ± 2.9**	47.68 ± 0.98	33.88 ± 1.35
crawler-url-parser	13.7 ± 5.4	9.0 ± 5.6	**29.4 ± 24.0**	**19.8 ± 16.1**	0.0 ± 0.0	0.0 ± 0.0
dropzone	1.3 ± 0.0	**1.4 ± 0.0**	**3.4 ± 0.0**	0.3 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
hoodie	**41.9 ± 1.8**	**31.4 ± 2.0**	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
node-dir	32.4 ± 0.4	22.2 ± 0.4	**58.7 ± 0.4**	**45.5 ± 0.6**	0.0 ± 0.0	0.0 ± 0.0
node-graceful-fs	17.5 ± 0.1	14.9 ± 0.1	**47.0 ± 1.0**	**33.0 ± 1.1**	40.04 ± 0.77	25.97 ± 0.77
postcss	36.4 ± 4.2	23.0 ± 4.1	**54.7 ± 0.3**	**40.0 ± 0.4**	44.59 ± 0.98	29.86 ± 1.09
supertest	25.7 ± 2.4	4.6 ± 2.1	**66.6 ± 4.7**	**52.3 ± 3.3**	3.31 ± 6.63	0.0 ± 0.0
Median	27.9	11.2	45.4	27.9	32.6	5.14

Table 6.4: Average statement and branch coverage (mean ± standard deviation over 10 runs) for SynTest, TestPilot_4omini, and TestPilot_4omini_baseline on selected benchmark projects.

Figure 6.4 provides a broader view of tool performance across all five metrics cal-

culated. TestPilot shows higher variation of coverage values but with noticeably higher average and median values in all structural coverage metrics. In contrast, SynTest shows a wider spread in pass rate values but much more consistent coverage values. The baseline LLM TestPilot generally produces coverage results similar to, but slightly lower than SynTest on average, while its coverage achievements are also more inconsistent, displaying a broader range of values. The distribution of function and line coverage mirrors trends seen in statement and branch coverage.

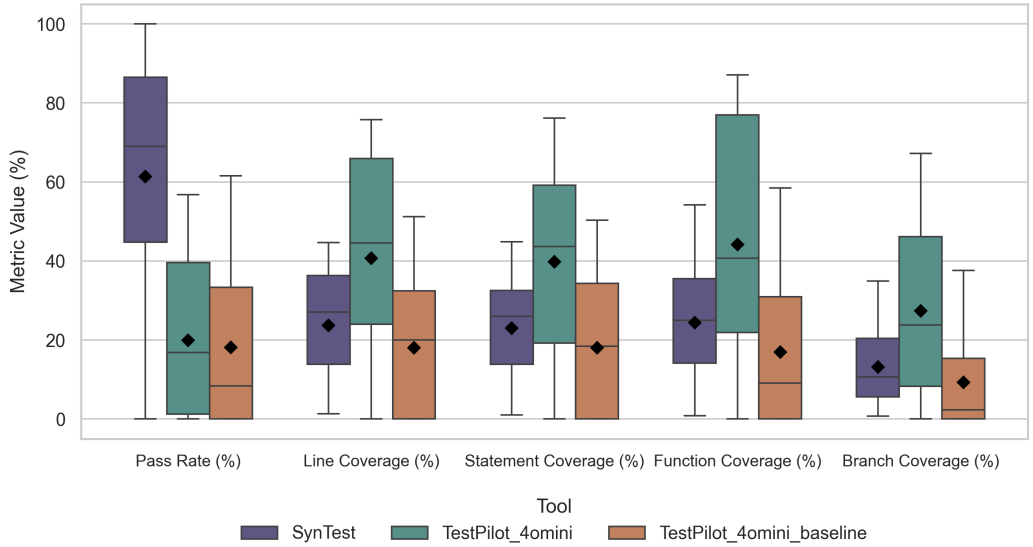


Figure 6.4: Test generation tool comparison across 10 runs of each project in the benchmark. Metrics include pass rate, line coverage, statement coverage, function coverage, and branch coverage. The diamond corresponds to the average value.

Aggregated results of the pairwise statistical comparisons, shown in Tables 6.5, 6.6, 6.7, summarize how many times each tool outperformed another across all projects for the key metrics of pass rate, statement coverage, and branch coverage.

The competition outcomes support the overall conclusions presented above: While SynTest more frequently generates tests that pass, TestPilot generally produces tests that cover much more of the code, demonstrating significant improvements over its zero-shot baseline version. The contrast is consistent across most projects in the benchmark and suggests differing strengths in how each tool approaches test generation.

	SynTest	**TestPilot_4omini**	**TestPilot_4omini_baseline**
SynTest	-	8	7
TestPilot_4omini	2	-	4
TestPilot_4omini_baseline	0	5	-

Table 6.5: Win counts between tools for pass rate across projects. Each cell shows how many projects the tool in the row significantly outperformed the tool in the column.

	SynTest	TestPilot_4omini	TestPilot_4omini_baseline
SynTest	-	1	5
TestPilot_4omini	8	-	9
TestPilot_4omini_baseline	4	0	-

Table 6.6: Win counts between tools for statement coverage across projects.

	SynTest	TestPilot_4omini	TestPilot_4omini_baseline
SynTest	-	2	6
TestPilot_4omini	7	-	9
TestPilot_4omini_baseline	4	0	-

Table 6.7: Win counts between tools for branch coverage across projects.

6.2.3 Test Examples

To better understand the performance results, we examined examples of generated tests from projects that exhibited poor performance or showed large differences between the tools.

hoodie

A significant discrepancy in coverage results is observed between SynTest and TestPilot: SynTest achieves an average coverage of 31.4%, whereas TestPilot reaches 0%. Examining the detected files and units reveals that TestPilot's API discovery process differs notably from SynTest's, leading to fewer identified targets. While SynTest detects exported units in 18 different files, TestPilot detects an exported unit in only one.

The common file, `hoodie/server/index.js`, contains a `register` function that accepts parameters of complex types:

```
module.exports.register = register
module.exports.register.attributes = {
  name: 'hoodie'
}

var corsHeaders = require('hapi-cors-headers')
var hoodieServer = require('@hoodie/server').register
var _ = require('lodash')

var registerPlugins = require('./plugins')

function register (server, options, next) {
  options = _.cloneDeep(options)
  _.defaultsDeep(options, {
    paths: {
      public: 'public'
    },
    plugins: [],
    app: {}
  })

  server.ext('onPreResponse', corsHeaders)
```

```
  server.register({ register: hoodieServer, options: options }, function
      (error) {
    if (error) {
      return next(error)
    }

    registerPlugins(server, options, function (error) {
      if (error) {
        return next(error)
      }

      next(null, server, options)
    })
  })
}
```

The following are tests generated by TestPilot for the given function.

```
let mocha = require('mocha');
let assert = require('assert');
let hoodie = require('hoodie');
describe('test suite', function() {
    it('test case', function(done) {
        let server = {}; // Mock server object
        let options = {}; // Mock options object

        hoodie.register(server, options, function(err) {
            assert.ifError(err);
            // Additional assertions can be added here
            done();
        });
    })
})
```

```
let mocha = require('mocha');
let assert = require('assert');
let hoodie = require('hoodie');
describe('test suite', function() {
    it('test case', function(done) {
        let mocha = require('mocha');
let assert = require('assert');
let hoodie = require('hoodie');

describe('test hoodie', function() {
    it('test hoodie.register', function(done) {
        let server = {
            ext: function() {}
        }; // Mock server object with ext function
        let options = {}; // Mock options object

        hoodie.register(server, options, function(error, registeredServer
            , registeredOptions) {
            assert.strictEqual(error, null);
            assert.strictEqual(registeredServer, server);
```

```
            assert.deepStrictEqual(registeredOptions, options);
            done();
        });
    });
});
    })
})
```

The first test fails due to the error: `TypeError: server.ext is not a function`. A subsequent retry modifies the mock object to include `server.ext` as a function, but this version fails to execute because of its format.

SynTest generates the following test:

```
// Imports
require = require('esm')(module)

describe('SynTest Test Suite', function() {
  let  register1;
  beforeEach(() => {
    // This is a hack to force the require cache to be emptied
    // Without this we would be using the same required object for each
        test
    delete require.cache[require.resolve("../../custom_benchmark/hoodie/
        server/index.js")];
    ({register: register1} = require("../../custom_benchmark/hoodie/
        server/index.js"));
  });

  it("Test 1", async () => {
    // Meta information
    // Selected for:
    //
    //
    // Covers objective: error:::d41d8cd98f00b204e9800998ecf8427e

    // Test
    const ext = () => {};
    const register = () => {};
    const server = {
      "ext": ext,
      "register": register
    }
    const options = true;
    const next = () => {};
    const registerReturnValue = await register1(server, options, next)
    const ext1 = () => {};
    const register2 = () => {};
    const server1 = {
      "ext": ext1,
      "register": register2
    }
    const options1 = true;
    const next1 = () => {};
    const anon = "util";
```

```
    const registerReturnValue1 = await register1(server1, options1, next1
        , anon)
    const ext2 = () => {};
    const register3 = () => {};
    const server2 = {
      "ext": ext2,
      "register": register3
    }
    const options2 = true;
    const ext3 = () => {};
    const registerReturnValue2 = await register1(server2, options2, ext3)

  })
})
```

This test runs successfully and correctly provides `server.ext`, resulting in higher coverage for SynTest. However, it contains no assertions, raising questions about its utility. Given the complexity of the `register` function, it is not surprising that both tools face challenges in producing meaningful tests for this unit.

supertest

A substantial discrepancy in coverage results is also observed between SynTest and TestPilot in the project supertest. SynTest achieves an average coverage of 4.6%, whereas TestPilot reaches 52.3%. The file `index.js` shows a large difference in coverage but it contains the following unit:

```
'use strict';

/**
 * Module dependencies.
 */
const methods = require('methods');
let http2;
try {
  http2 = require('http2'); // eslint-disable-line global-require
} catch (_) {
  // eslint-disable-line no-empty
}
const Test = require('./lib/test.js');
const agent = require('./lib/agent.js');

/**
 * Test against the given `app`,
 * returning a new `Test`.
 *
 * @param {Function|Server|String} app
 * @return {Test}
 * @api public
 */
module.exports = function(app, options = {}) {
  const obj = {};
```

```
  if (typeof app === 'function') {
    if (options.http2) {
      if (!http2) {
        throw new Error(
          'supertest: this version of Node.js does not support http2'
        );
      }
    }
  }

  methods.forEach(function(method) {
    obj[method] = function(url) {
      var test = new Test(app, method, url, options.http2);
      if (options.http2) {
        test.http2();
      }
      return test;
    };
  });

  // Support previous use of del
  obj.del = obj.delete;

  return obj;
};
```

One example test produced by TestPilot to target this unit is:

```
let mocha = require('mocha');
let assert = require('assert');
let supertest = require('supertest');
describe('test suite', function() {
    it('test case', function(done) {
        supertest(app)
            .get('/your-endpoint')
            .expect(200)
            .end(function(err, res) {
                if (err) return done(err);
                assert.strictEqual(res.body.message, 'Success');
                done();
            });
    })
})
```

This test fails due to `app` not being defined.

while SynTest produces this output to target the unit:

```
// Imports
require = require('esm')(module)

describe('SynTest Test Suite', function() {
  let anonymousFunction;
  let Test;
  beforeEach(() => {
```

```
    // This is a hack to force the require cache to be emptied
    // Without this we would be using the same required object for each
        test
    delete require.cache[require.resolve("../../custom_benchmark/
        supertest/index.js")];
    delete require.cache[require.resolve("../../custom_benchmark/
        supertest/lib/test.js")];
    (anonymousFunction = require("../../custom_benchmark/supertest/index.
        js"));
    (Test = require("../../custom_benchmark/supertest/lib/test.js"));
  });

  it("Test 1", async () => {
    // Meta information
    // Selected for:
    //
    //
    // Covers objective: error:::d41d8cd98f00b204e9800998ecf8427e

    // Test
    const app = -1;
    const http = true;
    const assignment = {
      "http2": http
    }
    const anonymousFunctionReturnValue = await anonymousFunction(app,
        assignment)
    const app1 = "$AS!OjlQdpQ)L|~J_-^@|bSz6gsW+]y'\\,HDK(1q/vXkzUoL}";
    const app2 = 2;
    const http1 = true;
    const assignment1 = {
      "http2": http1
    }
    const path = 2;
    const http2 = true;
    const assignment2 = new Test(app2, assignment1, path, http2)
    const anonymousFunctionReturnValue1 = await anonymousFunction(app1,
        assignment2)
    const app3 = 2;
    const http3 = true;
    const assignment3 = {
      "http2": http3
    }
    const path1 = 2;
    const http4 = true;
    const assignment4 = new Test(app3, assignment3, path1, http4)
    const http5 = true;
    const assignment5 = {
      "http2": http5
    }
    const http6 = true;
    const assignment6 = {
      "http2": http6
    }
```

```
    const assertHeaderReturnValue = await assignment4._assertHeader(
        assignment5, assignment6)

  })
})
```

While this test runs without errors, SynTest does not manage to construct the parameter objects as expected, limiting its coverage potential. In contrast, in several other generated tests, TestPilot successfully creates valid instances for the parameters required by the target function, leading to higher coverage.

For example:

```
let mocha = require('mocha');
let assert = require('assert');
let supertest = require('supertest');
describe('test suite', function() {
    it('test case', function(done) {
        const request = require('supertest');
        const express = require('express');

        const app = express();

        app.get('/test', function(req, res) {
            res.status(200).json({ message: 'success' });
        });

        request(app)
            .get('/test')
            .expect('Content-Type', /json/)
            .expect(200)
            .end(function(err, res) {
                if (err) return done(err);
                assert.strictEqual(res.body.message, 'success');
                done();
            });
    })
})
```

However, in this case the generated test does not actually target the intended unit. The prompt used by TestPilot refers to a different function (e.g., `Test.prototype.get`), meaning the high coverage is incidental rather than the result of deliberate targeting.

6.2.4 Failure Analysis

Figure 6.5 shows the distribution of failure types across all projects for both tools, highlighting which types of errors occur most frequently. The chart presents the proportion of each error category relative to the total number of failed tests per tool. As shown, both SynTest and TestPilot produce failures in all categories, but with different distributions. SynTest produces significantly fewer syntax and timeout errors compared to TestPilot. Instead, its failures are predominantly `AssertionError` and `TypeError`. Both tools produce very few file system errors, module not found errors and reference errors.

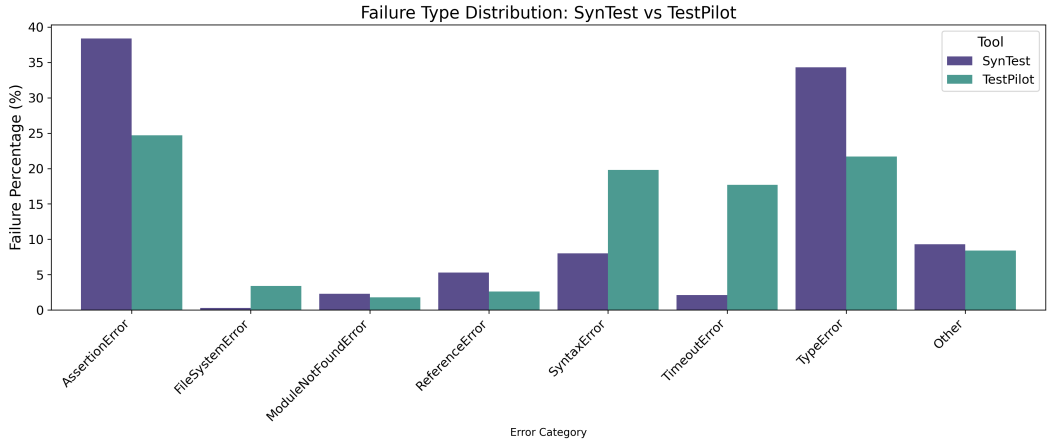


Figure 6.5: Comparison of failure type distributions for SynTest and TestPilot.

Figure 6.6 reveals that the types of failures differ greatly between projects, with each project exhibiting a unique distribution of error categories. While for some projects SynTest and TestPilot produce similar errors, many others show very different failure profiles between the tools. Interestingly, SynTest produces errors of only one type in several projects, namely dropzone, node-dir, node-graceful-fs, and supertest, with each of these exhibiting a different error type. This demonstrates that not only do error types vary widely per project, but the tools also differ in which failures they predominantly produce depending on the project context.

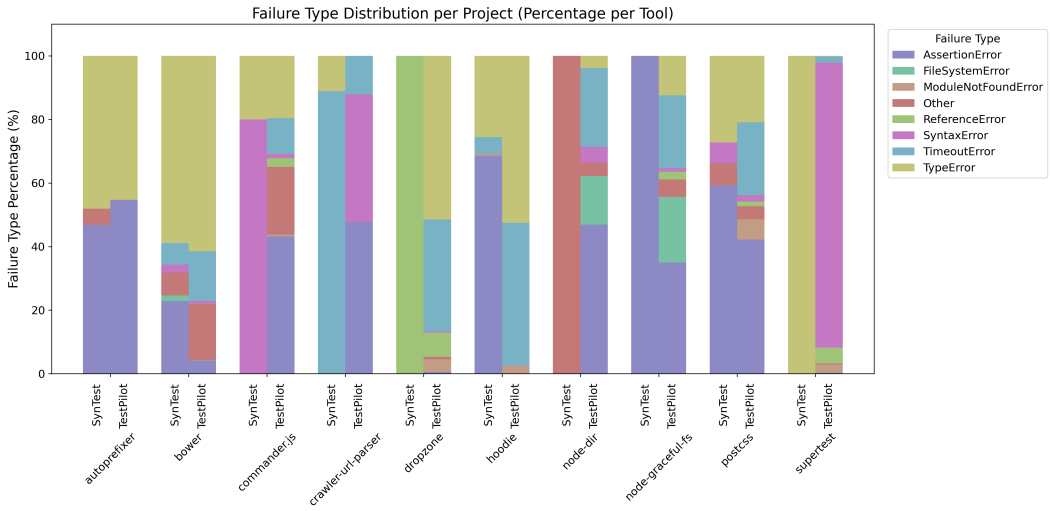


Figure 6.6: Comparison of failure type distributions per project.

6.2.5 Similarity Analysis

To better understand the degree of coverage overlap between the test cases generated by SynTest and TestPilot, Table 6.8 presents two Jaccard similarity (IoU) metrics. The *average Jaccard similarity* is computed by comparing the statements covered in each run of SynTest with each run of TestPilot, calculating the Jaccard Similarity for each pair and finding its average. This value is relatively low (0.29), indicating that the tools often produce test suites that cover different parts of the code.

In contrast, the *global Jaccard similarity* (0.60), is computed over the union of all statements covered across all runs. A venn diagram of this can be found in Figure 6.7. This higher value reflects that the tools do eventually cover many of the same statements, even if not consistently across all runs.

Additionally, two containment metrics are presented to show the extent to which one tool's coverage is covered by the other. On average, 80.95% of statements covered by SynTest are also covered by TestPilot, while only 31.45% of TestPilot's coverage is present in SynTest. This further reinforces the observation that each tool produces unique test cases, with TestPilot however covering a much broader range of statements.

Metric Type	Description	Value	Std Dev
Average Jaccard Similarity	Mean over all run-pair comparisons	0.2909	0.0885
Global Jaccard Similarity	Union-based similarity across all runs	0.6008	–
SynTest in TestPilot (%)	Mean proportion of SynTest statements also covered by TestPilot	80.95	13.63
TestPilot in SynTest (%)	Mean proportion of TestPilot statements also covered by SynTest	31.45	10.47

Table 6.8: Comparison of Statement Coverage Similarity Between Tools

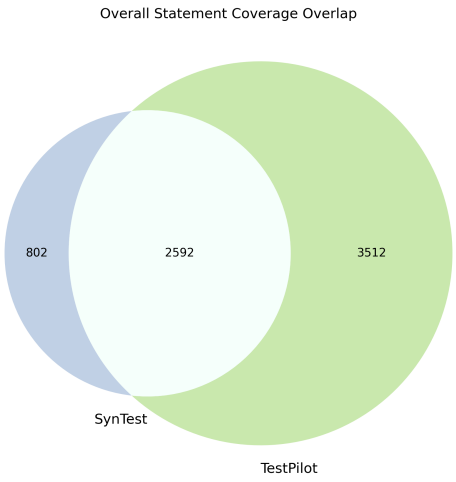


Figure 6.7: Venn diagram of accumulated statements covered by SynTest and TestPilot across all runs.

When looking at the per-project similarity, similar results can be found. While, values differ between projects, with the highest similarity observed for bower and the lowest for dropzone, the overall trend remains consistent across projects.

Project	Jaccard	syntest in testpilot (%)	testpilot in syntest (%)
autoprefixer	0.3158	92.05	32.46
bower	**0.3827**	67.81	**46.80**
commander.js	0.1473	73.18	15.56
crawler-url-parser	0.2348	78.31	25.56
dropzone	0.0000	0.00	0.00
node-dir	0.2445	80.49	26.00
node-graceful-fs	0.3096	86.99	32.47
postcss	0.3528	83.98	37.86
supertest	0.3495	**92.70**	36.01

Table 6.9: Per-Project Comparison with highest values for each metric in bold.

Findings: Tool compatibility issues limited the evaluation to a subset of projects, exposing gaps in the tools' generalizability. TestPilot produces significantly greater coverage than SynTest at the cost of large test suites and lower pass rates. The generated tests differ in quality and targeting precision, with TestPilot sometimes covering unintended code paths and SynTest producing tests that lack assertions. Overall, both tools show complementary strengths, exploring different code and exhibiting different failure patterns, underscoring the importance of diverse approaches in test generation.

6.3 Correlation Analysis

The third research question, as stated in Chapter 5, is:

> **RQ3:** How do tool performance outcomes correlate to code complexity characteristics and JavaScript features?

6.3.1 Project Level Results

The complexity correlation analysis at the project level reveals several notable trends. As shown in Figures 6.8a and 6.8b, SynTest shows a stronger negative correlation between code complexity and coverage metrics compared to TestPilot. Specifically, SynTest's performance tends to decline with increasing cyclomatic complexity and average number of features per project.

In contrast, TestPilot appears less affected by cyclomatic complexity, but its coverage decreases moderately as the average number of features increases. Interestingly, the number of units per project does not correlate with decreased coverage for SynTest, whereas for TestPilot, a larger number of units is associated with lower coverage. The total number of branches, however, does not exhibit a strong correlation with coverage for either tool.

The correlation between feature diversity and tool performance shows distinct differences between SynTest and TestPilot. As shown in Figures 6.9a and 6.9b, the tools respond very differently to the presence of specific JavaScript features across projects.

(a) SynTest: Coverage vs Complexity

(b) TestPilot: Coverage vs Complexity

Figure 6.8: Comparison of statement and branch coverage correlation with complexity features for SynTest and TestPilot on the project level.

For SynTest, negative correlations with coverage are observed for features such as `domInteraction`, `variadicParams`, and `objectManipulation`, suggesting that these constructs may introduce challenges during test generation. On the other hand, features like `async`, `closures`, and `propertyAccess` exhibit positive correlations with coverage, indicating that SynTest handles these more effectively.

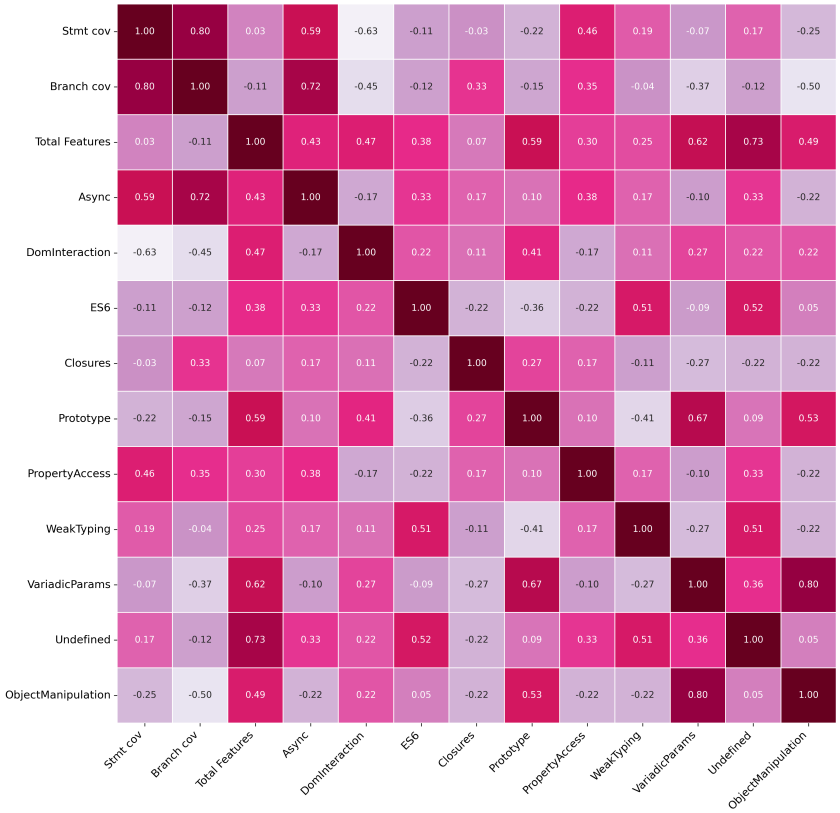
In contrast, TestPilot shows a broader negative correlation pattern: coverage tends to decrease with the overall number of features in a project, and with the presence of most individual features, except for `objectManipulation`, which shows a positive correlation. This may suggest that TestPilot struggles with highly feature-rich codebases, though it performs relatively well on complex projects.

These differences highlight that each tool has unique strengths and limitations when interacting with the diversity of JavaScript constructs.

6.3.2 File Level Results

At the file level, some different patterns of correlation between complexity metrics and coverage emerge for SynTest and TestPilot. Here, SynTest and TestPilot show very similar correlations over complexity metrics. Figures 6.10a and 6.10b illustrate how each tool's performance varies with complexity metrics across individual files.

For SynTest, no meaningful correlation is observed between coverage and cyclomatic complexity or the number of units in a file. However, both the average number of features per unit and the total number of branches show a negative correlation with coverage, suggesting that files with diverse features and complex branching logic pose greater challenges for SynTest.



(a) SynTest: Coverage vs Diversity

(b) TestPilot: Coverage vs Diversity

Figure 6.9: Comparison of statement and branch coverage correlation with diversity features for SynTest and TestPilot on the project level.

In contrast, TestPilot demonstrates a broader sensitivity to complexity metrics on the file level. While the number of units does not correlate with performance, coverage tends to decrease with increasing cyclomatic complexity, average features per unit, and total branches. This indicates that TestPilot's coverage is more strongly affected by complexity at the file level than SynTest's.

These results suggest that while both tools struggle with increasing feature richness and branching, TestPilot is more consistently impacted by complexity across different metrics.

(a) SynTest: Coverage vs Complexity

(b) TestPilot: Coverage vs Complexity

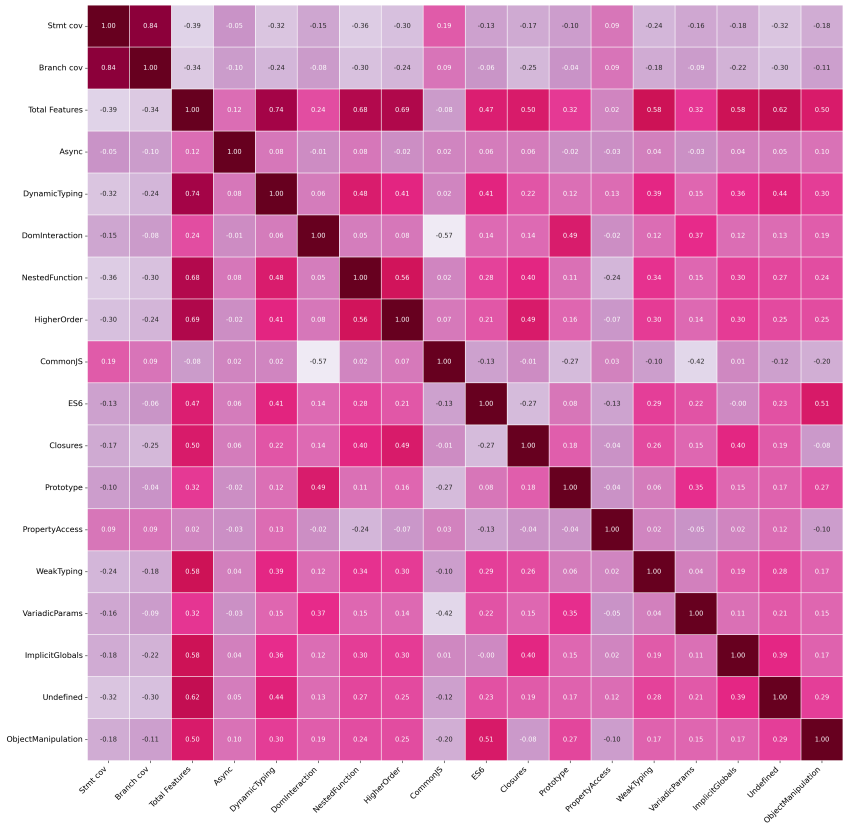
Figure 6.10: Comparison of statement and branch coverage correlation with complexity features for SynTest and TestPilot on the file level.

Again, compared to the project-level results, some file-level correlation values diverge. However, as can be seen in Figures 6.11a and 6.11b compared to 6.9a and 6.9b, overall, the colorings of the heatmaps are similar, showing that specifically, TestPilot shows very similar correlations.

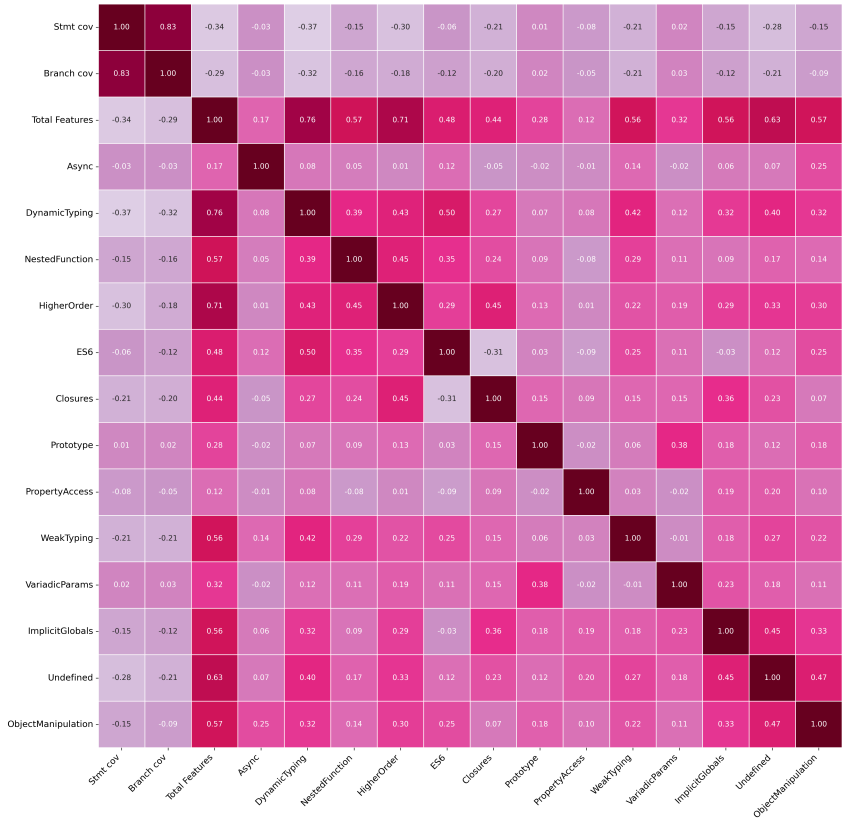
SynTest shows more negative correlations between coverage and features than on the project level. Specifically, regarding the total number of features, `nestedFunction`, `dynamicTyping`, `higherOrder`, `closures`, `implicitGlobals`, and `undefinedValues`. Only `commonJS` and `propertyAccess` show a slightly positive correlation.

For TestPilot, most features correlate negatively with coverage, with the exception of `async` and `prototype`, which show no significant relationship. This suggests that TestPilot may struggle more broadly with diverse feature presence at the file level, possibly due to the compounded complexity of combining multiple constructs within small, self-contained units.

Findings: Tool performance correlates differently with code complexity and JavaScript feature diversity, as well as on the file level compared to project level. SynTest's coverage generally decreases with higher complexity and certain features but is less impacted by project size, while TestPilot shows moderate sensitivity to complexity and a broader negative correlation with feature richness. Overall, both tools exhibit distinct weaknesses when handling diverse and complex JavaScript-specific constructs, highlighting the challenges that these language intricacies pose for automated test generation.



(a) SynTest: Coverage vs Diversity



(b) TestPilot: Coverage vs Diversity

Figure 6.11: Comparison of statement and branch coverage correlation with diversity features for SynTest and TestPilot on the file level.

Chapter 7

Discussion

This chapter discusses the main findings of this study in light of the three research questions outlined in Chapter 5. The goal is to interpret the results from Chapter 6 and identify any limitations that might affect their generalization. Each section below corresponds to one of the research questions and highlights key insights, challenges, and takeaways from the analysis. The chapter concludes with threats to validity.

7.1 RQ1: Existing Benchmark Evaluation

While SynTest and TestPilot benchmarks include relevant JavaScript features, both suffer from imbalanced feature representation and complexity. The comparison highlights a fundamental trade-off: small, carefully selected benchmarks (e.g., SynTest) offer depth but may struggle with representativeness, while large, unfiltered ones (e.g., TestPilot) achieve breadth but may lack complexity.

To understand why these imbalances occur, it's important to recognize that neither benchmark was designed to ensure balanced coverage of both features and complexity, a limitation that often arises from practical collection constraints and is likely not unique to these two benchmarks. Manual benchmark design is time-consuming, and the pool of well-structured, testable projects is limited. For JavaScript, achieving diversity is particularly difficult due to its diverse ecosystem. Automatic project collection can address this but risks bias toward popular frameworks or coding styles, while manual selection improves control but limits diversity and can introduce collection bias. In practice however, this means tools evaluated solely on those benchmark might appear more capable than they are when applied to feature-rich or more complex codebases.

Our custom benchmark addresses these issues but comprehensive benchmarks like ours are essential for driving progress in automated test generation research, providing realistic evaluations that reveal both strengths and weaknesses of tools. It can serve as a common ground for consistent community-wide comparison and collaborative improvement. Periodic updates will be necessary however to keep pace with the evolving JavaScript ecosystem, including deprecations and emerging features.

From this evaluation, we conclude that effective benchmark design requires carefully

balancing breadth and depth and leveraging innovative collection methods. Ongoing adaptation to language and ecosystem changes is essential to maintain meaningful assessments and drive progress in automated test generation research.

7.2 RQ2: Tool Performance Comparison

7.2.1 Tool Executability

A significant limitation observed is the low execution success rate of both tools across the full benchmark. This limited compatibility underscores a key challenge: despite advances in test generation, many tools remain fragile when confronted with the complexity and diversity of real-world codebases. These failures are not rare edge cases but common, as shown by the large portion of the benchmark excluded due to tooling limitations. Such brittleness limits usability and highlights that robustness, alongside coverage performance, must become a core focus for future tool development.

To overcome these challenges, future test generation tools should adapt dynamically to each project's environment and code characteristics. Tools such as intelligent SWE agents could easily achieve this by utilizing a feedback-driven setup approach. This adaptability would help tools handle setup and parsing issues more robustly, improving compatibility with diverse projects.

7.2.2 Performance Comparison on Runnable Projects

Across the benchmark projects, both tools show varied performance, with TestPilot however, overall achieving significantly higher coverage despite lower pass rates compared to SynTest. This reflects its aggressive test generation strategy. Although it results in many failing tests, the high generation volume ensures that enough valid and passing tests remain to contribute significantly to coverage. The zero-shot baseline TestPilot without prompt refinements or execution feedback performs substantially worse, highlighting both the effectiveness of TestPilot's multi-step prompting strategy and the significant progress in LLM-based test generation since the earliest implementations.

Compared to results reported in their original papers, both tools perform worse on our benchmark (TestPilot: 27.9% median branch coverage vs. 52.8% in original paper; SynTest: 11.2% vs. 46.0%), with high variation across projects. This is likely due to the broader feature diversity and higher code complexity in our dataset, which better captures real-world challenges. However, we further explore potential links between coverage, code features, and complexity in our correlation analysis, as well as through detailed test case examples.

Interestingly, comparing our findings to studies that evaluate testing tools on different programming languages reveals contrasting trends. In our experiments, LLM-based tools outperform search-based testing, while in others, search-based methods consistently demonstrate better results than LLM approaches [4]. This highlights how tool effectiveness can vary significantly depending on the language and context.

For practitioners, the results show a trade-off when selecting test generation tools. SynTest may be more suitable in contexts where stable, reliably passing tests are required, for

example, in CI pipelines or when tests are not expected to be human-reviewed or maintained. In contrast, TestPilot prioritizes coverage and fault discovery, producing a substantially larger number of test cases, many of which fail. This is particularly suitable when the tool is not used in a fully automated way. Where the test suites can be human-reviewed, and failed test cases can be easily cleaned up. While this aggressive LLM-based generation approach contributes to high code coverage, it also comes at a higher operational cost, especially since the tool relies on cloud-based inference and may transmit code or metadata externally. This raises considerations not only around cost but also data privacy and infrastructure requirements.

7.2.3 Test Examples

The examples from the projects hoodie and supertest illustrate key challenges in automated test generation for complex, real-world units. SynTest's deep search algorithm can enable it to mock required objects to a certain degree. However, unsurprisingly it falls short when SynTest fails to generate an `express` app, as this requires a degree of domain-specific knowledge about expected data types and frameworks. Another notable limitation of SynTest's generated tests is the lack of assertions, which reduces their practical value despite coverage numbers.

TestPilot's tests tend to be more natural, utilizing its meta-level domain knowledge, but often fail for simple reasons. Furthermore, TestPilot sometimes achieves higher coverage by targeting related functions rather than the primary unit, which inflates coverage metrics without guaranteeing comprehensive testing of the intended code.

The chosen projects contain human-written tests that adequately cover the example units, demonstrating that these complexities, while challenging, are realistically testable by humans. This highlights the current gap between automatic methods and human expertise, where humans can more easily incorporate the necessary contextual knowledge.

Regarding readability, both tools produce tests with shortcomings. SynTest's tests often include non-descriptive variable names like `register2` and `server1`, while TestPilot uses more generic but type-appropriate names. Both lack explanatory comments and clear objectives, making it difficult to understand the purpose of individual tests or the reasoning behind test inputs. This absence of context further limits the maintainability and usefulness of the generated tests.

Finally, it is important to note that the coverage differences observed between SynTest and TestPilot in these examples do not have straightforward explanations. The projects were specifically selected because the tools' performance differed substantially, yet in most cases the reasons for these discrepancies are multifaceted. This complexity underscores the challenge of automated test generation and suggests that no single approach consistently outperforms the other across all scenarios.

7.2.4 Failure Analysis

A search-based tool like SynTest exhibiting test errors and failures is unusual and is not accounted for in its original paper. The errors are of various nature but mainly caused by

assertion failures, syntax issues and type errors. While assertion failures can also indicate bug-revealing tests that are actually desirable, distinguishing these from failures caused by incorrect test logic or invalid assumptions is challenging and often requires manual inspection. They could therefore point either to successful fault detection or to errors in the tools implementation and limitations in how it infers types and input constraints during test generation.

TestPilot's higher rates of syntax and timeout errors result from its main reliance on LLMs, without strong parsing or execution validation, leading to the generation of syntactically incorrect and poorly formatted code. This issue is a known challenge in LLM-driven test generation.

7.2.5 Similarity Analysis

The similarity analysis reveals that SynTest and TestPilot generate complementary test suites, each exploring somewhat different parts of the codebase. Even after an aggregation of 10 runs, the Jaccard similarity is not 1 and not all of a single tool's coverage is contained within the other. Meaning that the differences in coverage are not merely due to variability in individual runs. This suggests that each tool's test generation approach has a unique exploration pattern, producing test cases that target distinct code paths or behaviors.

This complementary behavior has important practical implications. Combining the outputs of both tools may produce a more comprehensive test suite. It also points to opportunities for hybrid approaches that integrate the strengths of both methods to maximize code exploration.

7.3 RQ3: Correlation Analysis

The correlation analysis reveals clear differences in how SynTest and TestPilot seemingly respond to code complexity and feature richness. These relationships provide insight into the potential types of code each tool is better suited for and shed light on potential limitations in their design.

SynTest is more brittle with increasing cyclomatic complexity and diverse JavaScript features, confirming that search-based strategies struggle to scale with intricate control flows and feature interactions. TestPilot, while less sensitive to complexity, still drops in performance as feature richness grows—expected for LLM-based generation, which can be tripped up by loosely typed, non-deterministic, or highly functional constructs. Feature-level patterns largely match these trends: SynTest handles async and closures better but struggles with DOM interactions and variadic parameters; TestPilot handles object manipulation better but falters with dynamic typing and higher-order functions.

TestPilot's coverage declines as the number of units increases, which is unexpected since it generates tests at the function level. A possible reason is that projects with many units often have heavier cross-file dependencies that LLM prompts don't fully capture. At the file level, correlations often reverse or strengthen: SynTest's negative correlation with complexity becomes sharper, and certain features (e.g., nested functions, implicit globals) reduce coverage more locally than at the project level. TestPilot shows more consistently

negative correlations across most features at the file level, suggesting that dense feature combinations in a single file are harder for LLM-based generation than project-level averages imply.

Overall, SynTest is more limited by structural complexity while TestPilot is more sensitive to feature richness. Neither is robust across all scenarios, highlighting their complementary strengths and weaknesses. Utilizing this information could lead to feature-aware tool orchestration, where code segments are pre-analyzed and assigned to the tool most likely to succeed, leveraging the strengths of each tool and possibly increasing coverage and robustness.

7.4 Threats to Validity

7.4.1 Internal Validity

For both SynTest and TestPilot, flaky or inconsistent runs posed a threat. TestPilot's LLM-based generation introduces nondeterminism even when rerun with identical inputs, and SynTest occasionally failed due to internal errors or produced non-runnable test files. To mitigate this, we repeated experiments across multiple runs and reported average values where appropriate. However, tool crashes, environment mismatches, and limitations in the tools' ability to handle certain inputs could still have skewed the coverage or pass rate metrics.

Additionally, the evaluation pipeline involved multiple steps, including static analysis, execution tracing, and test validation. While we took care to automate and verify each step, the complexity of this pipeline may have introduced hidden bugs or inconsistencies. We used manual inspection to validate a subset of results, but full end-to-end verification is infeasible.

7.4.2 External Validity

External validity concerns the extent to which findings generalize beyond the studied benchmark. We carefully curated the benchmark to cover a variety of popular JavaScript and TypeScript projects, but the portion of runnable projects by both tools is still limited in size and scope. In particular, the benchmark focuses primarily on npm packages with exported backend units, and does not focus on, for example, front-end frameworks which may behave differently. We required all projects used for the evaluation to be runnable and compatible with the tools' test harnesses, which may bias the final dataset used in the evaluations to the tools.

Additionally, TestPilot was only used with a limited model, most similar to the one used in its original paper, which is not the largest or most capable one available, meaning results may not reflect its maximum potential performance. The study also focused specifically on white-box unit test generation, where the source code is available and the goal is to generate tests for it, rather than black-box test generation from natural language descriptions or high-level specifications.

Furthermore, the test suite evaluation pipeline may not generalize to other test generation tools, particularly those using very different approaches (e.g., symbolic execution) or with broader language support. Similarly, the observed tool behaviors may change with future updates.

7.4.3 Construct Validity

Construct validity addresses whether the chosen metrics are good proxies for what we aim to measure, in this case, the effectiveness and generalizability of test generation tools.

While coverage metrics are widely used, they do not capture the semantic quality or fault-finding ability of generated tests. High coverage does not necessarily mean that meaningful assertions are being made. Similarly, pass rate can be misleading. A high pass rate may reflect well-formed tests, but may also result from overly generic assertions. Conversely, failing tests may still be valuable if they expose unexpected behavior or subtle edge cases. In both tools, some failing tests stemmed from genuine assertion failures, while others resulted from invalid inputs, which calls for interpretation.

A potential threat to construct validity relates to data leakage for LLM-based tools such as TestPilot. We did not explicitly analyse potential effects of data leakage in our evaluation, primarily because our benchmark, after filtering, contained very few GitLab or new projects, with GitLab projects considered free from training data influence. This is particularly true for the final subset of projects that both tools could run, which only includes one GitLab project. Moreover, TestPilot's original paper performed a similar analysis and found no performance difference between GitHub and GitLab projects. In practice, data leakage may have limited impact, it can be viewed as the model leveraging knowledge of typical code constructs, which aligns with the intended behavior of LLM-based test generation. Supporting this, the single GitLab project included in our final evaluation did not show worse performance for TestPilot compared to other runs or SynTests performance, suggesting that any potential leakage did not distort our results significantly.

Chapter 8

Conclusions and Future Work

This chapter summarizes the main contributions of the project, reflects on key findings from the empirical evaluation, and outlines potential directions for future work.

8.1 Contributions

This thesis has presented a comparative evaluation of two state-of-the-art automated test generation tools for JavaScript, SynTest, a search-based tool, and TestPilot, an LLM-based tool. The main contributions are as follows:

- A carefully curated and runnable benchmark of real-world JavaScript and TypeScript projects, covering highly complex units and diverse language features, designed to support future evaluations of automated test generation tools.
- A reusable automated test suite evaluation pipeline that collects coverage and pass rate metrics and performs correlation analysis to code complexity and feature characteristics at multiple granularity levels, enabling consistent comparison across different test generation tools.
- A modular static analyzer designed to extract units under test and analyze their language features and complexity, forming a reusable tool for JavaScript dataset evaluation.
- A detailed empirical study of the tools' performance, showing how their coverage, pass rates, and robustness vary across different project characteristics and code features.
- A discussion of practical considerations in integrating and using these tools in real-world workflows, including observations about flaky executions, compatibility issues, and general tool maturity.

8.2 Reflection

The evaluation revealed that both tools have distinct strengths and weaknesses. TestPilot consistently achieved higher coverage, particularly branch coverage, likely due to its aggressive generation strategy and use of execution feedback. However, this came at the cost of test stability and higher failure rates, with a substantial portion of generated tests either failing or being invalid. Moreover, the tool's reliance on LLM prompting and remote execution introduced practical constraints in reproducibility and latency.

SynTest, on the other hand, produced fewer but more stable test cases, typically achieving significantly higher pass rates. Its search-based strategy, however, showed sensitivity to unfamiliar JavaScript constructs due to its reliance on static analysis and limited parser support.

The correlation analysis provided additional insights into how code characteristics affect tool performance. SynTest was more sensitive to complexity at the file level, especially branching and feature-rich units, while TestPilot's performance degraded more steadily with increased feature diversity. However, results also showed that both tools exhibit high performance variance across projects and that the tools often achieved distinct coverage, suggesting complementary strengths that could be leveraged in combination for more comprehensive test suites.

Finally, this study also highlighted broader challenges in evaluating automated testing tools: from defining meaningful metrics to ensuring tool compatibility in complex ecosystems like JavaScript. Despite advances in SOTA tooling, there remains a significant gap between academic performance claims and real-world applicability.

8.3 Future Work

There are several promising directions to build on this work:

- **Improved test quality assessment:** Future evaluations could go beyond structural metrics by incorporating *mutation scores*, *fault detection rates*, or more *manual inspections* to better assess the semantic quality of generated tests. This would provide a more holistic view of tool effectiveness.
- **Model selection for LLM-based tools:** Exploring how more recent, improved language models, used by either TestPilot or another LLM-based tool, perform within the same benchmark and evaluation framework used in this study could provide valuable insights into the impact of model choice on test generation effectiveness and the current status of LLM-based test generation.
- **Cross-language benchmarks:** The benchmark creation strategy proposed in this study, emphasizing language feature representation and code complexity, can be adapted to develop benchmarks for other programming languages.
- **Tool augmentation or hybridization:** Combining the strengths of search-based and LLM-based approaches, or improving the tool methodology evaluated, by utilizing

the insights gained from this thesis, could offer a tool with improved performance and coverage.

8.4 Conclusions

This thesis explored the current status of JavaScript test generation by analyzing how two fundamentally different state-of-the-art test generation tools behave on real-world JavaScript codebases. The results show that while LLM-based tools like TestPilot can generate higher coverage, they often do so at the cost of test generation bloat, producing a large volume of test cases, many of which fail or contribute little to meaningful validation. Search-based tools like SynTest offer higher test stability but may struggle with complex or highly dynamic code. Neither tool universally outperforms the other, and both face challenges in achieving robust, high-quality test generation across diverse and complex codebases.

The findings reinforce the importance of evaluating testing tools in realistic settings and using multiple perspectives, evaluation metrics, and analysis to understand their behavior.

Bibliography

[1] Ieee spectrum top programming languages 2024, 2024. URL https://spectrum.ieee.org/top-programming-languages-2024. Accessed: 2025-02-03.

[2] Stack overflow 2024 developer survey, 2024. URL https://survey.stackoverflow.co/2024/technology/. Accessed: 2025-02-03.

[3] A. Abdullin and V. Itsykson. KEX: A Platform for Analysis of JVM Programs. *Information and Control Systems*, (1):30–43, 2022. Online. Available: http://www.i-us.ru/index.php/ius/article/view/15201.

[4] Azat Abdullin, Pouria Derakhshanfar, and Annibale Panichella. Test wars: A comparative study of sbst, symbolic execution, and llm-based approaches to unit test generation. 1 2025.

[5] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, USA, 1 edition, 2008. ISBN 0521880386.

[6] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, Antonia Bertolino, J. Jenny Li, and Hong Zhu. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86:1978–2001, 8 2013. ISSN 01641212. doi: 10.1016/j.jss.2013.02.061.

[7] Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. A survey of dynamic analysis and test generation for javascript. *ACM Computing Surveys*, 50:1–36, 9 2018. ISSN 0360-0300. doi: 10.1145/3106739.

[8] Andrea Arcuri and Lionel Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24:219–250, 5 2014. ISSN 09600833. doi: 10.1002/stvr.1486.

[9] Andrea Arcuri and Gordon Fraser. Parameter tuning or default values? an empirical investigation in search-based software engineering. *Empirical Software Engineering*, 18:594–623, 6 2013. ISSN 1382-3256. doi: 10.1007/s10664-013-9249-9.

[10] Ellen Arteca, Sebastian Harner, Michael Pradel, and Frank Tip. Nessie. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1494–1505. ACM, 5 2022. ISBN 9781450392211. doi: 10.1145/3510003.3510106.

[11] Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. A framework for automated testing of javascript web applications. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 571–580. ACM, 5 2011. ISBN 9781450304450. doi: 10.1145/1985793.1985871.

[12] Matteo Biagiola, Gianluca Ghislotti, and Paolo Tonella. Improving the readability of automatically generated tests using large language models. In *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*, page 162–173. IEEE, March 2025. doi: 10.1109/icst62969.2025.10989020. URL http://dx.doi.org/10.1109/ICST62969.2025.10989020.

[13] Justus Bogner and Manuel Merkel. To type or not to type? In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 658–669. ACM, 5 2022. ISBN 9781450393034. doi: 10.1145/3524842.3528454.

[14] Hudson Borges, Andre Hora, and Marco Tulio Valente. Understanding the factors that impact the popularity of github repositories. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 334–344. IEEE, 10 2016. ISBN 978-1-5090-3806-0. doi: 10.1109/ICSME.2016.31.

[15] Islem Bouzenia, Bajaj Piyush Krishan, and Michael Pradel. Dypybench: A benchmark of executable python software. *Proceedings of the ACM on Software Engineering*, 1: 338–358, 7 2024. ISSN 2994-970X. doi: 10.1145/3643742.

[16] 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*, pages 572–576. ACM, 7 2024. ISBN 9798400706585. doi: 10.1145/3663529.3663801.

[17] Douglas Crockford. *JavaScript: The Good Parts: The Good Parts*. " O'Reilly Media, Inc.", 2008.

[18] Pedro Delgado-Pérez, Aurora Ramírez, Kevin J. Valle-Gómez, Inmaculada Medina-Bulo, and José Raúl Romero. Interevo-tr: Interactive evolutionary test generation with readability assessment. *IEEE Transactions on Software Engineering*, 49:2580–2596, 4 2023. ISSN 0098-5589. doi: 10.1109/TSE.2022.3227418.

[19] Nicolas Erni, Mohammed Al-Ameen, Christian Birchler, Pouria Derakhshanfar, Stephan Lukasczyk, and Sebastiano Panichella. Sbft tool competition 2024 - python test case generation track. In *Proceedings of the 17th ACM/IEEE International Workshop on Search-Based and Fuzz Testing*, pages 37–40. ACM, 4 2024. ISBN 9798400705625. doi: 10.1145/3643659.3643930.

[20] Gordon Fraser and Andrea Arcuri. Evosuite. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419. ACM, 9 2011. ISBN 9781450304436. doi: 10.1145/2025113.2025179.

[21] Gordon Fraser and Andrea Arcuri. A large-scale evaluation of automated unit test generation using evosuite. *ACM Transactions on Software Engineering and Methodology*, 24:1–42, 12 2014. ISSN 1049-331X. doi: 10.1145/2685612.

[22] Gordon Fraser and Andrea Arcuri. 1600 faults in 100 projects: automatically finding faults while achieving high coverage with evosuite. *Empirical Software Engineering*, 20:611–639, 6 2015. ISSN 1382-3256. doi: 10.1007/s10664-013-9288-2.

[23] Alessio Gambi, Gunel Jahangirova, Vincenzo Riccio, and Fiorella Zampetti. Sbst tool competition 2022. In *Proceedings of the 15th Workshop on Search-Based Software Testing*, pages 25–32. ACM, 5 2022. ISBN 9781450393188. doi: 10.1145/3526072.3527538.

[24] Rahul Gopinath, Carlos Jensen, and Alex Groce. Code coverage for suite evaluation by developers. In *Proceedings of the 36th International Conference on Software Engineering*, pages 72–82. ACM, 5 2014. ISBN 9781450327565. doi: 10.1145/2568225.2568278.

[25] Giovanni Grano, Christoph Laaber, Annibale Panichella, and Sebastiano Panichella. Testing with fewer resources: An adaptive approach to performance-aware test case generation. 7 2019. doi: 10.1109/TSE.2019.2946773.

[26] Siqi Gu, Quanjun Zhang, Kecheng Li, Chunrong Fang, Fangyuan Tian, Liuchuan Zhu, Jianyi Zhou, and Zhenyu Chen. Testart: Improving llm-based unit testing via co-evolution of automated generation and repair iteration. 8 2024.

[27] Peter Gyimesi, Bela Vancsics, Andrea Stocco, Davood Mazinanian, Arpad Beszedes, Rudolf Ferenc, and Ali Mesbah. Bugsjs: a benchmark of javascript bugs. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 90–101. IEEE, 4 2019. ISBN 978-1-7281-1736-2. doi: 10.1109/ICST.2019.00019.

[28] Gunel Jahangirova and Valerio Terragni. Sbft tool competition 2023 - java test case generation track. In *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*, pages 61–64. IEEE, 5 2023. ISBN 979-8-3503-0182-3. doi: 10.1109/SBFT59156.2023.00025.

[29] Kush Jain and Claire Le Goues. Testforge: Feedback-driven, agentic test suite generation. 3 2025.

[30] Kush Jain, Gabriel Synnaeve, and Baptiste Rozière. Testgeneval: A real world unit test generation and test completion benchmark. 10 2024.

[31] René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440. ACM, 7 2014. ISBN 9781450326452. doi: 10.1145/2610384.2628055.

[32] Jeshua S. Kracht, Jacob Z. Petrovic, and Kristen R. Walcott-Justice. Empirically evaluating the quality of automatically generated and manually written test suites. In *2014 14th International Conference on Quality Software*, pages 256–265. IEEE, 10 2014. ISBN 978-1-4799-7198-5. doi: 10.1109/QSIC.2014.33.

[33] 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)*, pages 919–931. IEEE, 5 2023. ISBN 978-1-6654-5701-9. doi: 10.1109/ICSE48619.2023.00085.

[34] Guodong Li, Esben Andreasen, and Indradeep Ghosh. Symjs: automatic symbolic testing of javascript web applications. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 449–459. ACM, 11 2014. ISBN 9781450330565. doi: 10.1145/2635868.2635913.

[35] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. Fuzzing: State of the art. *IEEE Transactions on Reliability*, 67:1199–1218, 9 2018. ISSN 0018-9529. doi: 10.1109/TR.2018.2834476.

[36] Andrea Lops, Fedelucio Narducci, Azzurra Ragone, Michelantonio Trizio, and Claudio Bartolini. A system for automated unit test generation using large language models and assessment of generated test suites. 8 2024.

[37] Stephan Lukasczyk and Gordon Fraser. Pynguin: Automated unit test generation for python. 2 2022. doi: 10.1145/3510454.3516829.

[38] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. An empirical study of automated unit test generation for python. 11 2021.

[39] Alexandre Matton, Tom Sherborne, Dennis Aumiller, Elena Tommasone, Milad Alizadeh, Jingyi He, Raymond Ma, Maxime Voisin, Ellen Gilsenan-McMahon, and Matthias Gallé. On leakage of code generation evaluation datasets. *arXiv preprint arXiv:2407.07565*, 2024.

[40] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2:308–320, 12 1976. ISSN 0098-5589. doi: 10.1109/TSE.1976.233837.

[41] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. Jseft: Automated javascript unit test generation. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10. IEEE, 4 2015. ISBN 978-1-4799-7125-1. doi: 10.1109/ICST.2015.7102595.

[42] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. Wiley Publishing, 3rd edition, 2011. ISBN 1118031962.

[43] N. Nachar et al. The mann-whitney u: A test for assessing whether two independent samples come from the same distribution. *Tutorials in Quantitative Methods for Psychology*, 4(1):13–20, 2008.

[44] Raphael Noemmer and Roman Haas. *An Evaluation of Test Suite Minimization Techniques*, pages 51–66. 12 2019. ISBN 978-3-030-35509-8. doi: 10.1007/978-3-030-35510-4_4.

[45] OpenAI. Gpt-3.5 turbo — openai api, 2024. URL `https://platform.openai.com/docs/models/gpt-3.5-turbo`. Accessed: 2025-04-13.

[46] Wendkuuni C. Ouedraogo, Kader Kabore, Haoye Tian, Yewei Song, Anil Koyuncu, Jacques Klein, David Lo, and Tegawende F. Bissyande. Llms and prompting for unit test generation: A large-scale evaluation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 2464–2465. ACM, 10 2024. ISBN 9798400712487. doi: 10.1145/3691620.3695330.

[47] Annibale Panichella and Mitchell Olsthoorn. *Higher Fault Detection Through Novel Density Estimators in Unit Test Generation*, pages 18–32. 2024. doi: 10.1007/978-3-031-64573-0_2.

[48] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering*, 44:122–158, 2 2018. ISSN 0098-5589. doi: 10.1109/TSE.2017.2663435.

[49] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. A large scale empirical comparison of state-of-the-art search-based test case generators. *Information and Software Technology*, 104:236–256, 12 2018. ISSN 09505849. doi: 10.1016/j.infsof.2018.08.009.

[50] Annibale Panichella, Sebastiano Panichella, Gordon Fraser, Anand Ashok Sawant, and Vincent J. Hellendoorn. Test smells 20 years later: detectability, validity, and reliability. *Empirical Software Engineering*, 27:170, 12 2022. ISSN 1382-3256. doi: 10.1007/s10664-022-10207-5.

[51] 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)*, pages 20–27. IEEE, 5 2021. ISBN 978-1-6654-4571-9. doi: 10.1109/SBST52555.2021.00011.

[52] Mike Papadakis and Nicos Malevris. Automatic mutation test case generation via dynamic symbolic execution. In *2010 IEEE 21st International Symposium on Software Reliability Engineering*, pages 121–130. IEEE, 11 2010. ISBN 978-1-4244-9056-1. doi: 10.1109/ISSRE.2010.38.

[53] Juan Altmayer Pizzorno and Emery D. Berger. Coverup: Coverage-guided llm-based test generation. 3 2024.

[54] June Sallou, Thomas Durieux, and Annibale Panichella. Breaking the silence: the threats of using llms in software engineering. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, pages 102–106. ACM, 4 2024. ISBN 9798400705007. doi: 10.1145/3639476.3639764.

[55] Arkadii Sapozhnikov, Mitchell Olsthoorn, Annibale Panichella, Vladimir Kovalenko, and Pouria Derakhshanfar. Testspark: Intellij idea's ultimate test generation companion. 1 2024.

[56] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An empirical evaluation of using large language models for automated unit test generation. 2 2023.

[57] Philip Sedgwick. Spearman's rank correlation coefficient. *BMJ*, page g7327, 11 2014. ISSN 0959-8138. doi: 10.1136/bmj.g7327.

[58] Marija Selakovic, Michael Pradel, Rezwana Karim, and Frank Tip. Test generation for higher-order functions in dynamic languages. *Proceedings of the ACM on Programming Languages*, 2:1–27, 10 2018. ISSN 2475-1421. doi: 10.1145/3276531.

[59] Koushik Sen. Concolic testing. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 571–572. ACM, 11 2007. ISBN 9781595938824. doi: 10.1145/1321631.1321746.

[60] Domenico Serra, Giovanni Grano, Fabio Palomba, Filomena Ferrucci, Harald C. Gall, and Alberto Bacchelli. On the effectiveness of manual and automatic unit test generation: Ten years later. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 121–125. IEEE, 5 2019. ISBN 978-1-7281-3412-3. doi: 10.1109/MSR.2019.00028.

[61] S. S. Shapiro and M. B. Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52:591, 12 1965. ISSN 00063444. doi: 10.2307/2333709.

[62] 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*, pages 313–322. ACM, 6 2024. ISBN 9798400717017. doi: 10.1145/3661167.3661216.

[63] André Silva, Nuno Saavedra, and Martin Monperrus. Gitbug-java: A reproducible benchmark of recent java bugs. In *Proceedings of the 21st International Conference on Mining Software Repositories*, pages 118–122. ACM, 4 2024. ISBN 9798400705878. doi: 10.1145/3643991.3644884.

[64] M. Skoglund and P. Runeson. A case study on regression test suite maintenance in system evolution. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pages 438–442. IEEE. ISBN 0-7695-2213-0. doi: 10.1109/ICSM.2004.1357831.

[65] Marius Smytzek, Martin Eberlein, Batuhan Serçe, Lars Grunske, and Andreas Zeller. Tests4py: A benchmark for system testing. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 557–561. ACM, 7 2024. ISBN 9798400706585. doi: 10.1145/3663529.3663798.

[66] Mozhan Soltani, Annibale Panichella, and Arie van Deursen. Search-based crash reproduction and its impact on debugging. *IEEE Transactions on Software Engineering*, 46:1294–1317, 12 2020. ISSN 0098-5589. doi: 10.1109/TSE.2018.2877664.

[67] Dimitri Stallenberg, Mitchell Olsthoorn, and Annibale Panichella. *Guess What: Test Case Generation for Javascript with Unsupervised Probabilistic Type Inference*, pages 67–82. 2022. doi: 10.1007/978-3-031-21251-2_5.

[68] Yutian Tang, Zhijie Liu, Zhichao Zhou, and Xiapu Luo. Chatgpt vs sbst: A comparative assessment of unit test suite generation. 7 2023.

[69] Mubarak Albarka Umar. Comprehensive study of software testing: Categories, levels, techniques, and types, 6 2020.

[70] A. Vargha and H. D. Delaney. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[71] Kaixin Wang, Tianlin Li, Xiaoyu Zhang, Chong Wang, Weisong Sun, Yang Liu, and Bin Shi. Software development life cycle perspective: A survey of benchmarks for code large language models and agents. 2025. doi: 10.48550/ARXIV.2505.05283. URL https://arxiv.org/abs/2505.05283.

[72] Wenhan Wang, Chenyuan Yang, Zhijie Wang, Yuheng Huang, Zhaoyang Chu, Da Song, Lingming Zhang, An Ran Chen, and Lei Ma. Testeval: Benchmarking large language models for test case generation. 6 2024.

[73] Yibo Wang, Congying Xia, Wenting Zhao, Jiangshu Du, Chunyu Miao, Zhongfen Deng, Philip S. Yu, and Chen Xing. Projecttest: A project-level llm unit test generation benchmark and impact of error fixing mechanisms, 2025. URL https://arxiv.org/abs/2502.06556.

[74] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, Brian Goh, Ferdian Thung, Hong Jin Kang, Thong Hoang, David Lo, and Eng Lieh Ouh. Bugsinpy: A database of existing bugs in python programs to enable controlled testing and debugging studies. 1 2024. doi: 10.1145/3368089.3417943.

[75] Dinuka R. Wijendra and K.P. Hewagamage. Analysis of cognitive complexity with cyclomatic complexity metric of software. *International Journal of Computer Applications*, 174:14–19, 2 2021. ISSN 09758887. doi: 10.5120/ijca2021921066.

[76] Chen Yang, Junjie Chen, Bin Lin, Jianyi Zhou, and Ziqi Wang. Enhancing llm-based test generation for hard-to-cover branches via program analysis. 4 2024.

[77] Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. Evaluating and improving chatgpt for unit test generation. *Proceedings of the ACM on Software Engineering*, 1:1703–1726, 7 2024. ISSN 2994-970X. doi: 10.1145/3660783.

[78] Zhengran Zeng, Yidong Wang, Rui Xie, Wei Ye, and Shikun Zhang. Coderujb: An executable and unified java benchmark for practical programming scenarios, 2024. URL `https://arxiv.org/abs/2403.19287`.

[79] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: A survey for roadmap. *ACM Computing Surveys*, 54:1–36, 1 2022. ISSN 0360-0300. doi: 10.1145/3512345.