University of Twente

Agent-based Test Generation with Large Language Models and Stryker

Everard de Vree

Master's Thesis

Computer Science

Supervisors:
Arend Rensink, Fernando Castor de Lima Filho (University of Twente)
Jochem Groot Roessink, Rinse van Hees (Info Support)

Enschede, Overijssel
2026

ABSTRACT

Software Testing is an essential but time-intensive discipline within Software Engineering. Once a software test is in place, we can evaluate its quality using, among other metrics, mutation testing. This method of analysis uses artificial source code bugs (mutations), and measures how many of them cause test failure. It is itself notoriously time consuming, but provides feedback on the essential part of Software Testing: potential fault detection.

Recently, Large Language Models (LLMs) have shown promising results in automating a wide range of classical Software Engineering problems. Additionally, these models can be trained to use tools to form so-called *agents*, capable of completely autonomous work. In this thesis, we extend research in this direction by providing an agent with various tools before prompting it for the generation of entire JavaScript test suites. In order to provide feedback signals on generated test quality, we integrate the JavaScript mutation framework StrykerJS at various points in the test generation process.

We evaluate the effectiveness and cost of the agentic test generation approach by comparing against an LLM-based JavaScript test generation framework that integrates test quality feedback, but does not enable autonomous tool calls. Additionally, we measure the effectiveness of the mutation feedback by splitting our experiments into groups with and without detailed mutant information. We find that our solution can create stronger tests while reducing LLM token costs. Mutation analysis proves to be a useful signal, but only when scoped correctly.

CONTENTS

CHAPTER 1: INTRODUCTION

Large Language Models (LLMs) have taken the software engineering field by storm. Their generalization capabilities and impressive performance on generative coding tasks have prompted research into applying them in classic programming disciplines. One practice where LLMs are an attractive option is test case creation, a traditionally time-consuming endeavour for software developers: Daka et al. find that developers spend around 15.8% of their time writing unit tests.[14]

LLMs are neural language models: their design is inspired by the way human brains processes natural language. They consist of many small, interconnected computational units which are (greatly oversimplified) models of human neurons. In LLMs, billions of these computational units are organized into layers and trained on large bodies of natural language text to learn statistical patterns in language use. This allows the model to predict and generate text that is coherent and contextually appropriate, which has proven useful: LLMs have shown remarkable performance across a broad range of natural language tasks, including machine translation [63], text summarization [56], and question answering [17].

Before the advent of LLMs, Hindle et al. already observed that software shares many statistical regularities with natural language [29]. This allowed for joint natural language and source code modeling, which ultimately led to the powerful code-enabled LLMs we see today. Modern LLMs are trained on large corpora of both natural language and code, and can respond to natural language prompts with code completion, function generation, and reasoning about program behaviour [10].

1.1 Research Gap

As LLMs became capable of producing coherent and semantically plausible code, researchers started exploring whether they could assist with test creation, which can be labor-intensive. Typically, an LLM will be prompted with a function under test and tasked with generating a unit test. The resulting machine-generated tests are often high in readability and decent in code coverage scores, but they can contain hallucinated code or fail to compile [9]. These limitations, combined with increasing context windows that allow larger prompts, have led to the development of iterative test-generation approaches, in which the models are re-prompted with feedback extracted from previous attempts in order to progressively improve the quality and correctness of the generated tests.

Such feedback can take the form of execution and compilation results, or test-quality metrics such as code coverage and mutation scores. These iterative approaches report outperforming classical (non-LLM), as well as zero- and few-shot approaches. They find improved test correctness, coverage, and robustness compared to previous state-of-the-art [6], [11], [15], [27], [51], [70].

There is still room for improvement in feedback-driven approaches. Celik et al. [9] have surveyed LLM-based test generation and find several limitations in such systems. Invalid test cases are still generated, and test effectiveness takes a hit when programs depend on external resources or undocumented dependencies. Performance drops when documentation is ambiguous in general, and when projects are not modular or contain complex structures like inheritance. To mitigate these limitations, the authors recommend exploring hybrid approaches that combine LLM-based generation with additional sources of feedback and program understanding.

1.2 Approach and Research Questions

A natural response to the recommendations from the previous section is to combine LLM generation with existing, established software engineering tools. Our industrial partner Info Support, which maintains the mutation testing tool Stryker, is particularly interested in the effectiveness of mutation analysis as a feedback signal. Stryker, like other mutation testing frameworks, provides information about whether tests can catch artificial bugs. We select the JavaScript flavour of the project, StrykerJS, to develop our own hybrid test generation system and investigate the usefulness of this feedback during the generation process.

The feedback mechanisms used by iterative test generation frameworks depend on external compilers, test runners and static analyzers that each need their own programmatic integration into the framework. Modularity depends on the implementation details, and extending the loop with additional feedback will always require a software integration step. In our solution, we use a simple, recently developed interface between the language model and the tools that support it: The Model Context Protocol (MCP) [32].

MCP servers provide a well-defined interface that allows an LLM to call external tools directly, using its own judgement. This can simplify the construction of feedback loops: If we provide the right prompt and tools, the model makes its own decisions about when to obtain and act on feedback signals, reducing time spent programming the desired behaviour. Alongside a custom StrykerJS MCP server, we leverage the MCP interface to provide the agent with access to a code search tool for gathering context, and access to the filesystem for reading and writing code.

We provide mutation feedback at two points in the test generation process. First, the model gets access to fine-grained mutation feedback per API item exposed by the library under test. Then, after tests have been generated for all items, the agent is provided with project-wide information on which code blocks still produce a poor mutation analysis signal. This phase will be referred to as the *residual feedback phase*

This thesis aims to investigates whether such an MCP-based, feedback-driven architecture can improve JavaScript unit test generation, and in particular whether making mutation feedback visible to the model improves the quality of the generated tests. We formulate the following research questions:

RQ1: How effective is an MCP-based agentic test generation system for generating JavaScript unit tests?

RQ2: How does incorporating mutation testing feedback through StrykerJS affect the quality of LLM-generated tests?

1.3 Contributions

This thesis makes the following contributions:

- We design and implement an agent that generates test suites for JavaScript libraries. It implements LLM-based test generation with project exploration, source-code search, test execution, and StrykerJS-based mutation analysis through tool calls.
- We introduce a mutation-guided generation loop in which the agent can iteratively improve passing tests using StrykerJS feedback targeted per API item. This allows the model to simultaneously receive feedback on whether tests execute and whether they detect behavioural changes in the code under test.
- We add and evaluate a residual mutation feedback phase that gives the agent an additional opportunity to integrate mutation feedback not emitted by the regular public API generation loop.
- We empirically evaluate the system on a JavaScript library benchmark derived from another LLM-based test generator for JavaScript: TestPilot [55]. The evaluation compares hidden and visible mutation feedback conditions, analyzes the cost and benefit of the residual feedback phase, and compares the resulting tests against TestPilot in terms of correctness, mutation score, and token usage.

The remaining chapters of this document are organised as follows: Chapter 2 contains background information for the relevant technical concepts, Chapter 3 will provide an overview of the related work, and Chapter 4 details the research questions and methodology. Chapter 5 contains the implementation and design rationale. Finally, Chapter 6 reports the experimental results and Chapter 7 provides discussion and our conclusion.

CHAPTER 2: BACKGROUND

In this chapter, we provide relevant background information for the interested reader unfamiliar with some of the technical domains involved in the thesis.

2.1 Large Language Models

Large language models (LLMs) are a class of language models that excel in Natural Language Processing (NLP). As the name suggests, NLP concerns itself with machine-based processing of natural language. However, the capabilities of modern LLMs have grown far beyond the processing of natural languages. Many modern models support multimodal inputs (e.g. images, audio), code generation, and tool use.

2.1.1 History

In order to solve fundamental NLP tasks like speech recognition and natural language generation, current state-of-the-art LLMs have evolved significantly from their first ancestors in the 1990s, Statistical Language Models (SLMs) [66].

Statistical Language Models

Given a textual context, these models provide a probability distribution over the set of possible next words. This distribution is estimated by training on a large dataset that likely contains the current context. The frequency of all words following this context in the dataset is then normalized to obtain a probability, after which the next word with the highest probability is selected. Another name for SLMs is n-gram, referring to this generation of the n'th word for a context size of $n-1$. A severe limitation of n-grams is exponential growth of the required dataset with increased context size. In practice, this limited the number n of included words to 1 or 2 at the time [66]. Additionally, SLMs cannot generalize to n-grams that do not appear in the training data.

Neural Language Models

In 2003, Bengio et al. introduced their Neural Probabilistic Language Model, which was the first Neural Language Model (NLM) to outperform state-of-the-art SLMs [5]. NLMs rely on encoding words as large vectors of real numbers, representing points in a high-dimensional

numerical space. With enough dimensions, the position of a word in this space can capture aspects of its semantics. Words that are close to each other in this space are then likely to have a similar meaning, and vector operations can be performed to verify the encoding of semantics.

NLMs derive their name from the involvement of neural networks, which are computational models inspired by the human brain. They consist of many small units of computation called neurons. These neurons are stacked in layers which operate on the input signals of previous layers and gradually adjust the connection weight to different neurons during the training stage. The layered nature allows NLMs to extract different features of text at different levels of abstraction. The true semantics of these layers are difficult to analyze, but we can derive an intuitive example by assuming that the extracted features map nicely to grammar constructs. For example: one layer detects simple patterns like adjective-noun relationships, and another layer receives these patterns as inputs and finds out which verbs these adjective-noun combinations relate to. The last layer is the size of the model's vocabulary. Each neuron outputs a value between 0 and 1, representing the probability that its associated token is the next token in the sequence. The next token is then picked from the highest ranking outputs, with some degree of randomness.

The randomness in this sampling process is controlled by a parameter called *temperature*. Temperature modifies the probability distribution over possible next tokens before the final token is selected. A low temperature makes the distribution sharper, meaning that the model is more likely to select high-probability tokens and therefore produce more predictable output. A higher temperature flattens the distribution, which gives tokens of a lower probability a higher chance of being selected and making the output more varied, but also less reliable.

Recurrent Neural Networks & Transformers

The NLMs discussed so far handle each input independently: there is no feedback from one prediction to the next. It is possible to improve performance in tasks involving sequential data, by adding a feedback loop. That is in essence what a Recurrent Neural Network (RNN) is. Instead of only feeding information forward to the next layer, neurons in an RNN can feed their output to a hidden state. This hidden state is combined to the input at every time step, allowing RNNs to keep information "in memory" in between inputs. RNNs are very well-suited for processing sequential data like audio streams, and also perform well in machine translation [37].

However, RNNs struggle when connections between inputs are spread over a longer range, and their sequential processing cannot be parallelized. These limitations were overcome in

the Transformer architecture, proposed in 2017 by Vaswani et al. [63] The Transformer allows every token to interact with every other token in the input through a process called self-attention. Transformers also inject extra positional information to allow parallelization of this process without losing the order of the input tokens. As a result, the Transformer achieved state-of-the-art results on machine translation tasks while training faster and more efficiently. Since then, it has become the backbone of large language models, supporting further scaling and longer context windows.

Pre-trained Language Models

The increased scaling enabled by the Transformer architecture laid the foundation for the Pre-Trained Model (PTM) paradigm. Instead of being trained from scratch for the task at hand, training can be split into a general and specialized phase for large enough models.

In the first stage, pre-training, PTMs are supplied with a large corpus of data for self-supervised learning. For text generation, self-supervised learning looks like this: Given a complete sentence in the training data, iterate through the sentence word by word. For every iteration, provide only the context as input to the model, and set the next (known) word as a training target. This proved to be an effective way of providing language models with a basic "understanding" of NLP.

After the pre-training phase, PTMs can be adapted to specific tasks by means of so-called "fine-tuning", during which they undergo a final stage of supervised learning. A simple example is a PTM adapted to sentiment classification. In its final fine-tuning stage, it receives extra training on a dataset that is labeled by humans with labels like "positive" and "negative". In 2018, Devlin et al. brought the pre-trained paradigm to natural language processing, with the Transformer-based BERT model [17]. BERT surpassed state-of-the-art prior work in multiple NLP benchmarks, and supported simple fine-tuning for a variety of tasks.

Large Language Models

The success gained by large PTM models based on the Transformer architecture like BERT (its large version had 340 million parameters) prompted research into scaling them even further. It turns out PTMs scale remarkably well, and parameter counts and training data size grew quickly. More funding and research led to the development of models with billions of parameter. By 2020, Brown et al. presented GPT-3 with 175 billion parameters [7]. Their model demonstrated that massive scale and self-supervised pre-training enables strong performance, without requiring any fine-tuning. Models of such scale became known as Large

Language Models, named after the large numbers of parameters used in their neural networks and the massive datasets they are trained on. They have proven useful in a variety of research fields including medicine, law, finance, and education [66].

2.1.2 Cost

LLM providers typically bill usage based on the number of tokens processed by the models. One token does not necessarily translate to one word or character, but refers to a unit of text produced by a tokenizer. A tokenizer might convert entire words to single tokens, but often words are split into multiple tokens. The cost of an LLM request is therefore dependent on the size of the natural language context (often a query) that a user provides, called a prompt, and the number of output tokens. However, it is not always possible to precisely calculate the cost of a prompt beforehand, since some providers keep their tokenizers private.

Most LLM providers also distinguish between input and output tokens, and generally charge more per output token. This happens because during the input processing phase, the model can process the previously mentioned self-attention over the prompt in parallel. When the model starts outputting tokens, each next token depends on the previous output. This means that the output phase is sequential across tokens, which keeps resources occupied for a longer time.

There is a third category: cached input tokens. When the same prompt prefix is reused across requests, e.g. a system prompt with tool instructions, providers can cache the input after processing and charge a lower rate for later re-use of those tokens. We can see the proportions in the pricing of some contemporary models available in Table 2.1.

Table 2.1: Example API token prices for selected LLM providers

Provider	Model	Input	Cached input	Output
OpenAI [47]	GPT-5.5	$5.00	$0.50	$30.00
Anthropic [3]	Claude Sonnet 4.6	$3.00	$0.30	$15.00
Anthropic [3]	Claude Haiku 4.5	$1.00	$0.10	$5.00
Google [26]	Gemini 3.5 Flash	$1.50	$0.15	$9.00

Prices are shown in USD per one million tokens.

Trends

The cost of LLM inference has generally decreased when measured relative to model capability: over time, achieving similar levels of benchmark performance has become cheaper [13],

[58]. However, this does not mean that newly released models are cheaper than their predecessors. Frontier and reasoning-oriented models can be priced substantially higher than smaller or older models. Additionally, they often have larger context windows or longer responses, which can further increase the total cost of a task. For example, GPT-4.5 was introduced after GPT-4o and much more expensive: input prices increased from $2.50 to $75 per million and output tokens went from $10 to $150 per million [45].

It should be noted that these numbers are publicly listed API prices instead of the actual cost for providers when serving a request. The real cost of inference depends on many different physical factors like the operational expenses for data centers alongside business considerations. As a result, listed prices cannot be interpreted as a direct measurement of the cost of inference. They are still relevant for the of API users, because they determine the actual cost of using these models.

2.1.3 Prompt Engineering

Because natural language is inherently flexible, the prompts that language models process can be expressed in many different forms, and small differences in phrasing can affect the quality of an LLM's output [8]. The search for optimized prompts became its own discipline: *prompt engineering*. However, the internal "reasoning" processes of LLMs are poorly traceable and interpretable due to very large size of the models and their computational complexity. This makes prompt engineering a process of mostly trial and error, sensitive to model-specific behaviour. Still, some general techniques have been published.

Zero- and Few-shot prompting

Zero-shot prompting is a simple prompting technique, which only requires writing natural language instructions or task descriptions. A zero-shot prompt contains no examples, unlike the *few-shot prompting* technique proposed by Brown et al. in 2020 [7]. The authors' strategy consists of providing a few solved tasks alongside the actual task description. They demonstrate that this improves the performance of GPT-3 considerably, performing comparably to fine-tuned models across a variety of benchmarks. However, in 2021, Reynolds et al. have found simple zero-shot prompts that can outperform few-shot prompts [52]. They explain that examples do not always help because the LLM might misinterpret them as part of the task description, instead of providing general guidance. In a 2023 survey on zero-prompting, Li. et al. argue that examples in few-shot prompting lead to biased predictions [38]. This could decrease performance for the unseen tasks that differ from the examples too much. These mixed results highlight the fragile nature of prompt engineering: comparing results

across the large number of ways to phrase a task in natural language is difficult. Especially for LLMs whose internal semantics are hard to trace.

Chain-of-Thought

In 2022 Wei et al. proposed Chain-of-Thought prompting, which involves providing step-by-step solutions for the example problems included in few-shot prompts [67]. After performing experiments on various reasoning benchmarks, the authors conclude that their relatively simple technique improves reasoning capability for sufficiently large LLMs. That same year, Kojima. et al demonstrate that a zero-shot version of this technique also leads to significant gains in reasoning performance over simpler zero-shot techniques [34]. This can be achieved by simply appending "Let's think step by step" to a zero-shot prompt.

2.1.4 LLMs for Code

LLMs are not limited to interpreting and generating natural languages. Provided with sufficient training, they can operate on programming languages just as well. An example is OpenAI's Codex family of GPT models, fine-tuned on GitHub code. In 2021, Chen et al. describe the training process of the first Codex model and evaluate its performance [10]. It outperformed contemporary GPT models at code generation tasks, but suffered some limitations like struggling with correctly generating long chains of operations.

Still, code completion and generation became some of the most popular and commercially successful applications of LLMs. A notable example is the Codex-powered Visual Studio Code extension GitHub Copilot. It was introduced as a preview in June 2021, before GitHub made the tool generally available to individual developers in June 2022 [21], [24]. Its rapid adoption suggests that code completion and generation became one of the first commercially significant applications of LLMs in software engineering. For example, Microsoft reported in early 2024 that GitHub Copilot had more than 1.3 million paid subscribers and was used by more than 50,000 organizations [41]. As new developments in LLMs research improved code generation, applications shifted to more complex generation tasks, including automatic generation of test suites.

Challenges

When prompting LLMs for complex code-related tasks, the context window (maximum size of the prompt + output) limits the analysis of complex programs. This can cause LLM-based test generators to struggle with generating sensible tests for programs that depend on multiple internal modules.

A related challenge is dependency understanding on the repository level. In 2025, Du et al. introduced DependEval, a benchmark for evaluating LLMs on dependency recognition, repository construction, and file editing on real-world repositories [18]. Their results show substantial performance gaps on tasks that require models to infer dependencies and maintain consistency across files.

2.2 AI agents

When LLMs are provided with tools they can control autonomously, they become *AI agents*. Although some authors make an explicit distinction between *agent-based* and *agentic* systems (the latter referring only to interconnected multi-agent systems [54]), we will use the word *agentic* to ergonomically refer to autonomous, tool-using behaviour. Recently, LLMs have proven to be effective drivers of agentic workflows after various LLM providers started supporting external function calling and tool use. American AI research organization OpenAI pioneered external function calling in the API for its GPT models [22]. Soon after, the organization developed ChatGPT plugins, which enable developers to integrate their external tools with the UI of OpenAI's chatbot. In the following period, numerous other providers like Google [22] and Microsoft [23] launched custom tool and plugin interfaces for their LLMs. Enabling external tool functionality usually involves an extra training step that teaches the models to output special tokens to request an available tool. The introduction of a variety of such interfaces with different syntax fragmented the development landscape [30]

2.2.1 Model Context Protocol

In 2024, the American AI company Anthropic (known for developing the LLM family Claude) released the Model Context Protocol [32] in an effort to unify engineering efforts. It is an open standard for registration, exploration, and execution of tools for AI agents. Inspired by the Language Server Protocol [44], which standardizes communication between IDEs with language intelligence services, the standard contains specifications for hosts, servers, and clients. The host is the AI application itself, often in the form of a chatbot or an Interactive Development Environment. External tools and resources are available on dedicated MCP servers. When a host needs access to an MCP server, it launches an MCP client that connects with the server and discovers the available tools. The MCP server advertises its *tools* (endpoints and usage schemas), *resources* (data) and pre-defined *prompts*. This information is then provided to the host, after which the agents may begin using the available resources.

Besides the standard output of a tool call or resource request, the MCP server can send

notifications. The MCP client can communicate these back to the host, to keep the AI application up to date with e.g. progress updates. Another interesting innovation of MCP is the possibility of bidirectional communication: If the host allows it, an MCP server may *sample* the host AI application for additional information during a tool call.

2.3 Mutation Testing

Mutation testing is a powerful tool in Software Testing that allows developers to test the effectiveness of a test suite by following a straightforward process. First, the syntax of the program under test is mutated some way that is intended to modify its semantics. The next step is detecting whether the test suite detects this "mutation" and fails, or if the mutation survives and all tests succeed. In order to derive useful conclusions from these outcomes, two important assumptions must be made. The first is that programmers generally make programs that do not differ greatly from their intended functionality (Competent Programmer Hypothesis). This narrows the scope of mutation analysis down to mutants that are close to the correct program. The second assumption is that test cases sensitive enough to "kill" simple faults (i.e. fail on a mutated program) are likely sensitive enough to kill more complex faults. The combined effect of these hypotheses is powerful: by generating artificial, simple mutations, we can simulate real and complex bugs.

The results of mutation analysis are useful in practice: Instead of waiting for real faults to appear or manually reasoning about the thoroughness of a test suite, developers can obtain a quantitative measure of test adequacy. Mutation testing can highlight untested behaviours, weak assertions, and missing edge cases.

Although the core idea is simple, mutation testing remains an area of active research more than 50 years after it was first suggested by Lipton in a student report in 1971 [39].

Mutation Operators

Mutation testing works by introducing changes to source code using a set of possible mutations defined as *mutation operators*. For example, an arithmetic mutation operator might replace a `+` sign with a `-`, while a relational operator mutation could change `>` to `>=`.

Most mutation testing frameworks allow users to select which mutation operators to apply. This flexibility enables trade-offs between thoroughness and computational cost. Introducing more operators will produce a larger number of mutants, increasing the time required for analysis but potentially revealing new weaknesses. In practice, developers often adjust these settings depending on available resources: for example, running a more exhaustive

mutation analysis during periodic quality assessments, and using a smaller operator set for faster, continuous integration feedback.

2.3.1 Metrics & Mutant Equivalence

A commonly used mutation-based metric for test suites is *mutation score*, which is defined as the ratio of killed mutants to generated mutants. A low mutation score means that many injected faults (mutants) survived the tests, which indicates weak fault-detecting ability. A high mutation score means that most mutants were killed, showing that the test suite is effective at catching bugs.

The accuracy of mutation score as a test effectiveness metric is subject to one problem: equivalent mutants. An equivalent mutant is a generated mutant that is semantically equivalent to the original program. Executing a test suite on such a mutant is a waste of resources, and can make the resulting mutation score less reliable: An equivalent mutant is impossible to kill, since it will always return the same outputs for all test inputs. Generating equivalent mutants will therefore result in a pessimistic estimate of the true mutation score. However, finding out if two programs are equivalent is a fundamentally undecidable problem. As a result, many tools include them in the mutation score unless they are manually inspected or removed by additional analysis. Mutation testing tools do often automatically discard mutants that are syntactically invalid, fail to compile, or cannot be executed due to runtime or configuration errors.

Another equivalence problem occurs when two mutants are functionally identical to each other, but different from the source program. These so-called redundant mutants will always be killed by the same test cases, so generating and evaluating them again is a waste of resources. Additionally, killing a redundant mutant skews the mutation score more than it should, because it is increased twice for an equivalent mutation. The opposite holds for a surviving redundant mutant.

State-of-the-art Equivalent Mutant Detection

Equivalent mutants are a practical problem: the rate at which they are produced varies with programming language, mutation operators, and programs, but empirical studies commonly find non-trivial proportions. For example, Kushigian et al. manually analysed 1,992 mutants from seven open-source Java projects and identified 215 equivalent mutants, with a median equivalent-mutant rate of 2.97% [35]. Reducing equivalent mutant generation rates would improve the usefulness of mutation scores and potentially boost the adoption of mutation testing, which is why it is an area of active research. A wide variety of algorithms, heuristics,

and verification methods has been introduced to the mutation testing community throughout the years, and recent LLM-based equivalent mutant detection has shown promising results (see Section 3.3).

2.3.2 Strong and Weak Mutation Testing

A more accurate name for the process described so far would be *strong* mutation testing. This refers to the case where mutations are used to validate test oracles (i.e. checks) specifically: Given a (non-equivalently) mutated PUT and our input test data, do our assertions and checks kill this mutant? The process is powerful, but executing the test program fully for all test inputs can be prohibitively expensive for many larger software systems.

In 1982 Howden et al. proposed weak mutation testing [31] to mitigate this problem. Weak mutation testing uses more relaxed criteria for the killing of mutants, meaning a weak mutation test does not require a different output in order to kill a mutant. Instead, there is only one condition: Is there a test input with which we reach a different state right after the mutated statement? Intuitively, this means that we are no longer validating the test oracles, but checking that the test inputs reach the mutated state, and that the test is "sensitive" to the mutated code. On its own, this does not prove anything, but the absence of such a sensitivity means that there is a potential gap in the test data.

Example

A classic example to illustrate the otherwise abstract difference between weak and strong mutation testing, is presented below. Listing 2.1 contains the original function, while Listing 2.2 shows the same function with a mutated comparison operator.

In this case, the mutation changes the relational operator from `>` to `>=`. Assuming numeric inputs, this mutation is equivalent under strong mutation testing: there is no input for which the original and mutated functions return different values. When `a > b`, both versions return `a`; when `a < b`, both versions return `b`; and when `a == b`, the original takes the false branch while the mutant takes the true branch, but both branches return the same value.

However, the mutant is not equivalent under weak mutation testing. For the input `max(3, 3)`, the original and mutated programs take different branches immediately after evaluating the conditional. This creates an internal state difference, even though the difference does not propagate to the output. A test suite consisting only of `assertEquals(max(3,3), 3)` would therefore be deemed acceptable by weak mutation testing, but not by strong mutation testing. The mutant is weakly killed, but it cannot be strongly killed.

In practice, weak mutation testing sacrifices accuracy for a significant performance boost.

```
export function max(a, b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

Listing 2.1: Original function

```
export function max(a, b) {
  if (a >= b) {
    return a;
  } else {
    return b;
  }
}
```

Listing 2.2: Mutated function

It has recently seen use for validating deep learning models, but industrial adoption is still limited. For the remainder of this thesis, we will only discuss strong mutation testing, as that is the kind that Stryker provides.

2.3.3 Mutation Testing With Stryker

Stryker Mutator is an open source framework for strong mutation testing with support for JavaScript, C# and Scala [59]. Its architecture is the same across all flavours, though some features are not feasible or implemented for all languages. Recent work describes strykerJS as a state-of-the-art mutation testing tool for JavaScript [62]. Stryker.NET is also a recognized mutation testing tool in its ecosystem, and is used as the example tool in Microsoft's official documentation on mutation testing for .NET [42].

The rest of this work will focus on StrykerJS, the JavaScript flavour of the framework. It supports most many major JavaScript and TypeScript test runners through a plugin system, together with a generic solution for other cases. In this thesis, we will use the Mocha runner, which runs tests through the widely used JavaScript test framework Mocha.

Dry Run

After StrykerJS mutates the target source code, it performs a so-called "dry-run". During this phase, Stryker simply executes the tests suite with mutants disabled, and aborts if any tests fail. It is a relatively cheap way to identify configuration errors or broken tests before spending computation on the expensive actual mutation testing.

Nondeterminism

Multiple StrykerJS runs on the same project can produce different mutation scores. One cause is that an inconsistent test suite will naturally produce inconsistent mutation result:

a flaky test that happens to pass the dry run and then fail during mutation testing will kill mutants that might survive in another run.

StrykerJS classifies the results of mutation testing per mutant as killed/timed out/survived/no coverage. The timeout classification also leads to a variance in mutation score. Reducing the timeout threshold can increase the number of timeouts and vice versa. Even with the same threshold between runs, small differences in scheduling and resource use can make a difference.

Reports

After executing a mutation test, Stryker provides a detailed report alongside the mutation score and error rates. The mutation report contains the modified source code of each mutant and an indication of whether it was killed, and by which test. There is a variety of reporting formats available, including a remote dashboard and real-time output. If it becomes clear that additional context may be helpful, Stryker's modular architecture allows for the creation of custom (reporter) plugins.

CHAPTER 3: RELATED WORK

Mutation testing is not a new technology, but LLM-based test generation has only recently entered the software engineering research space. Using mutation score as feedback for this test generation is an even newer direction, but it has already produced some promising research. This chapter will give an overview of related work in these areas, and identify research gaps.

3.1 LLMs for Software Testing

Given the time-consuming nature of writing software tests, it is no surprise that there is plenty of research available for offloading tasks in this domain to LLMs. In one of the first large surveys of studies researching LLM-enabled software testing, Wang et al. identify 102 papers (dated up to October 2023) from 4 databases based on relevancy and quality [65]. They find that major categories of LLM application in this domain are test case preparation, test oracle generation, and program repair. We limit our exploration to test case preparation, i.e. generating test inputs and execution steps, and test oracle generation: judging whether the displayed behaviour is correct, e.g. through assertions. We will treat both as elements of test case generation in order to simplify cross-referencing with other papers.

3.1.1 Surveys of the Landscape

Wang et al. survey LLM use in software testing and report that test-case preparation is among the most studied tasks [65]. They note that about a third of reviewed studies use pre-training or fine-tuning, while prompt-based approaches are dominated by zero-shot and few-shot prompting; more advanced strategies such as chain-of-thought are rarely used. They identify improved oracle handling, more rigorous evaluation, advanced prompting, and tighter integration with traditional testing techniques as important future directions. In particular, they discuss the use of accompanying techniques alongside LLMs, which motivates later work that combines LLM prompting with static analysis, dynamic analysis, or execution feedback.

Many of their conclusions and recommendations are consolidated by Zhang et al. in their 2025 survey on the use of LLMs, specifically for unit testing [71]. They describe a shift in the field from model adaptation toward prompt- and context-based techniques. Earlier studies often adapted models to testing tasks through pre-training, fine-tuning, or reinforcement

learning, partly because then-available models had weaker out-of-the-box code-generation capabilities. Later studies more often rely on stronger general-purpose or code-oriented LLMs and focus on improving the information provided to the model, for example through prompt engineering, retrieval, or feedback from test execution. Despite this progress, Zhang et al. report that generated tests still suffer from compilation errors, incorrect assertions, and inconsistent evaluation standards. They therefore recommend combining LLMs with software-engineering techniques such as static analysis, dynamic analysis, and execution feedback.

Celik et al. provide an even more recent perspective in their review on test case generation, which includes papers until July 2025 [9]. They identify similar challenges: compilation errors and test oracles that cannot distinguish correct from incorrect behaviour are still prominent obstacles. In addition, they heavily emphasize the need for comprehensive metrics and universal benchmarks.

Utilized LLMs

Celik et al. report that the majority of the LLMs used in test generation research belong to the OpenAI GPT family [9]. The popular models GPT-3.5 and GPT-4 were available early, and have strong zero-shot learning capabilities. Additionally, they have a rich, extensively documented ecosystem compared to smaller or open-source models. Still, many researchers include open-source models like LLaMA, CodeLLaMA and DeepSeek Coder to improve transparency and reproducibility.

Godage et al. conduct a comparison study including GPT-4o, Claude 3.5, Command-r-plus-08-2024, and Llama 3.1 [25]. The generated tests are measured by test success rate, statement coverage, and mutation score. Claude 3.5 ends up as the top performer in all categories.

These findings are in line with the results of the LiveBench benchmark introduced by White et al. in 2024 [68]. LiveBench consists of new benchmarks for six LLM task categories including coding. The questions are refreshed every 6 months and the authors claim that the questions are uncontaminated (i.e. not seen in any LLM training data). claude-3-5-sonnet-20240620 (a version of Claude Sonnet 3.5 released in June of 2024) outperforms 39 other LLMs in the coding category and scores second best overall, just behind OpenAI o1.

Metrics & Benchmarks

Zhang et al. report that among LLM-based unit test generation studies, code coverage is the most commonly used metric, followed by pass rate and compile rate [71]. Coverage tools are

available in many ecosystems and show whether generated tests execute the source code, but do not indicate whether their assertions actually check for the intended behavior. This is why mutation score is a useful complementary metric: instead of measuring only execution, it measures whether the test suite detects actual behavioral changes in the program. Table 3.1 summarizes the metrics from Zhang et al. that are most relevant to test generation from source code alone, where no reference implementation or formal specification is available.

Table 3.1: Selected metrics for LLM-based unit test generation

Metric	**Count**	**Description**
Code Coverage	38	The percentage of code (e.g., lines, branches) executed by a test suite. Indicates how much of the system is explored by the generated tests. Useful for measuring exploration of the code under test, but not assertion quality.
Pass Rate	27	The proportion of generated test cases that execute without failing, and have no failing checks (assertions). This is only an executability metric: A high value does not necessarily imply that the tests catch faults in the system.
Compile Rate	21	The percentage of generated tests that successfully compile. This is a stricter criterion than syntactic validity, because syntactically valid tests may still fail due to type errors, missing imports, unavailable symbols, or incorrect use of the test framework. For interpreted languages, comparable measures include import success, parse success, or execution success.
Syntax Rate	14	The proportion of generated test code that is syntactically valid. A weaker executability metric than compile rate: syntactically valid code may still fail to compile or execute.
Mutation Score	9	The proportion of non-equivalent mutants killed by the generated test suite (see Section 2.3.1).
Bug Detection	8	The ability of tests or generated code to reveal faults. Tested on datasets with known, ground truth bugs.
Cyclomatic Complexity	2	The number of independent control-flow paths through code. It is framed as a maintainability or readability metric for generated tests, not as a measure of fault-detection power.

Note. The counts refer to the number of papers, out of 105 papers reviewed by Zhang et al., that report each metric [71]. The counts are not mutually exclusive, since a single paper may report multiple metrics. The following metrics were omitted because they require a predetermined "correct answer" for comparison, which is not available when new tests are generated from source code alone: Correct Rate, Syntactic Accuracy, BLEU, and CodeBLEU.

One issue that the authors report is that many studies only evaluate a very limited selection of metrics. Zhang et al. [71] report that most test generation studies rely on code

coverage, pass rate, and compile rate, while only eight examine whether generated tests uncover real-world bugs. One case only reports the number of generated tests [53].

Another significant problem is the lack of standardized benchmarking. In order to make a meaningful comparison across different test generators, it is essential to evaluate them on the same input programs. However, the datasets used in practice vary wildly: Zhang et al. [71] attempt a performance evaluation but find that all included papers benchmark against a different dataset, so no cross-comparison can be done.

3.2 Techniques for Enhancing LLM-Based Test Generation

The black-box nature of LLMs turns prompt engineering (discussed in Section 2.1.3) into more of an art than a science. Slightly modified wording in a prompt might have a significant effect on LLM output, but the reasons might be hard to trace. Still, providing extra context does generally improve the quality of outputs, which can be empirically verified. Roy Chowdhury et al. [53] present a simple example; they find an increased number of generated tests when augmenting their prompt with type information and method signatures.

This idea can be expanded across different types of context, or even providing dynamic feedback in loops. Several recent LLM-based unit-test generators therefore combine prompting with additional static context, execution feedback, repair loops, coverage guidance, or mutation feedback. Zhang et al. [71] see this as part of a broader trend toward hybrid LLM-based unit-testing techniques. This section provides a deep dive with concrete examples of such contemporary test generators.

3.2.1 Mutation-guided Test Generation

As shown in Table 3.1, many LLM-based test generation techniques discussed in the investigations in Section 3.1.1 use metrics such as line or branch coverage. However, coverage alone is not a complete metric for test effectiveness: a test suite may achieve high coverage yet still detect very few actual faults. For instance, recent work shows that some generated test suites achieve 100% line coverage yet only 4% mutation score [64].

The computational cost of mutation analysis remains a drawback, but it is less problematic when used for one-time use for test generation instead of continuous test execution. Moreover, LLM-based test generation can imply a context where extra computation is accepted as the cost of automation: LLMs facilitate a high degree of automation, but have relatively high latency and compute costs. Still, the stronger justification is that mutation score captures a property that coverage does not: whether generated tests are capable of catching behavioral changes in the code under test.

MuTAP

One of the firsts report of an LLM-based test generation system guided by mutation analysis was MuTAP, released by Dakhel et al. in July 2024 [15]. The system takes a python program under test as input together with an LLM and the type of initial prompt (zero- or few-shot). The LLM is prompted with an initial prompt, and the resulting unit test is

syntax-checked and refined. syntax errors are fed back into the LLM, and all assertions in the test are updated to expect the actual values produced by the test inputs. Then MuTAP runs the Python mutation testing tool MutPy. All surviving mutants are returned in an "augmentation prompt" that tells the model to add a test killing the mutant. MuTAP iterates while there are surviving mutants that have not yet been used for prompt augmentation stops when all mutants are killed or when no unused surviving mutant remains. Finally, MuTAP applies a minimization algorithm to retain the smallest set of assertions that preserves the achieved mutation score.

The results are encouraging: on HumanEval, the best MuTAP configuration achieves a mutation score of 93.57%, killing 1179 out of 1260 mutants. This is a significant increase from 65.94% and 649 killed mutants for Pynguin, a fully automated unit test generation framework for Python, which uses traditional test-generation techniques to produce regression tests with high code coverage [40]. It also reaches a perfect mutation score for 114 out of 164 PUTs, compared with 46 for Pynguin. On the Refactory benchmark of real buggy student submissions, the same configuration detects 94.91% of bugs, compared with 67.54% for Pynguin [15]. These results make MuTAP a useful example of mutation analysis being used not only as an evaluation metric, but also as feedback for improving LLM-generated tests.

ACH

Meta engineers introduced a similar iterative mutation-guided system in February 2025, when they reported on successfully integrating a tool called ACH with their pipeline [19]. The engineers instruct an LLM to create mutants simulating real-world issues, and then prompt an LLM to generate a test in order to kill this mutant. Additionally, there is an LLM-enabled equivalent mutant detection step to improve resource efficiency. Overall, 73% of generated tests were useful enough to accept into the test suite, enough to prove the worth of this paradigm in industrial settings.

MutGen

Later in 2025, Wang et al. introduce an LLM-based framework for generating Java unit tests: MutGen [64]. MutGen iterates on its tests with feedback from PITest, also known as PIT, a widely used mutation testing system for Java and the JVM [49]. The feedback includes the mutation location, mutant status, and mutation operator. The system also includes a fixing step for repairing runtime errors, compilation errors, and assertion failures. Test generation is repeated iteratively: after each round, MutGen re-evaluates the current test suite, feeds

the remaining live and uncovered mutants back into the prompt, and continues until the mutation score converges or a predefined iteration limit is reached.

The evaluation shows a substantial improvement over the baselines. The main non-LLM baseline is EvoSuite, a search-based Java unit-test generation tool that automatically generates JUnit tests [20]. MutGen achieves mutation scores of 89.5% on HumanEval-Java and 89.1% on LeetCode-Java, compared with 69.5% and 58.9% for EvoSuite. Even when EvoSuite is configured as EvoSuite$_{mut}$ to optimize for mutation score, it reaches only 67.4% and 58.1% on the two benchmarks [64]. The authors also present the new dataset LeetCode-Java, extracted from public GitHub repositories containing Medium and Hard LeetCode solutions. The paper contains an ablation study in which code summarization, fixing, and mutation feedback are disabled to assess the contribution of each component. In MutGen, code summarization is a preprocessing step in which the LLM summarizes the behaviour of the method under test from its implementation, so that test generation is guided by a description of the actual code rather than only by existing comments or method names. The authors conclude that this summarization step has the largest impact, because original comments can mislead the LLM or omit details such as input formats; the fixing step also matters, especially on LeetCode-Java; and mutation feedback gives an additional, smaller improvement by helping the model target harder-to-kill mutants.

3.2.2 TestPilot

For JavaScript, the most directly relevant LLM-based unit test generation framework is TestPilot [55]. It targets JavaScript libraries distributed as npm packages, i.e., reusable Node.js packages that expose an API to client programs. For such a package under test, TestPilot first dynamically explores the package's exported object graph to identify exported API functions. For each discovered function, it records an access path, a signature with parameter names, and the function body.

TestPilot then generates tests for Mocha, a common JavaScript testing framework. The initial prompt contains the test scaffolding, imports, and the recorded access path and signature of the function under test. Additional prompts are then produced by adding different kinds of context. The available prompt refinements are:

- the function body of the method under test;
- documentation comments on the method under test;
- usage snippets mined from documentation;
- a failing test from a previous iteration, together with its error message.

These refinements are applied independently and in all possible combinations. When several kinds of context appear in the same prompt, TestPilot uses a fixed order: usage snippets, previous failure information, documentation comments, function body, and finally the function signature.

The authors evaluate TestPilot on 25 JavaScript packages from the npm ecosystem, covering 1,684 API functions. The first 10 packages are taken from the evaluation of Nessie, a feedback-directed JavaScript test generator for APIs with synchronous and asynchronous callbacks [4]. Because those packages mainly cover I/O-related, callback-heavy libraries, Schäfer et al. add 10 packages from other domains and programming styles, as well as 5 packages hosted on GitLab to reduce the risk that the evaluated code was present in the LLM's training data [55].

Compared with Nessie, TestPilot achieves higher median coverage: 70.2% statement coverage and 52.8% branch coverage, against 51.3% and 25.6% for Nessie. These differences are statistically significant in the authors' evaluation. There is another difference between the tools: Nessie generates tests only based on coverage feedback which end up without any assertions, whereas TestPilot often produces more "natural-looking" tests with assertions. The authors report that a median of 61.4% of generated tests contain *non-trivial assertions*. This is the authors' own proxy for measuring test oracle quality: the number of assertions which depend on at least one function from the package under test [55].

3.3 LLMs for Equivalent Mutant Detection and Mutant Generation

As mentioned in Section 2.3.1, equivalent mutants are a major challenge for mutation testing because they reduce the reliability of the mutation score and introduce unnecessary execution effort. Recent work has explored whether LLMs can help detect such mutants more accurately. In addition, LLMs have been investigated as a way to generate mutants that better resemble real-world bugs than traditional operator-based mutation tools.

Tian et al. claim to be the first to systematically study LLM-based strategies for equivalent mutant detection [61]. They evaluate several LLM-based techniques on 3,302 method-level Java mutant pairs and report substantially better scores than existing equivalent mutant detection methods. Their study also considers efficiency, measuring training and inference time, and concludes that LLM-based techniques can offer a favorable trade-off between detection effectiveness and the computational overhead of equivalent mutant detection.

On the mutant generation side, Tip et al. show that their LLMorpheus can generate

effective custom mutants using LLMs. These mutants are generated by an LLM from the surrounding code context, and then executed and classified using a modified version of StrykerJS [62]. This approach targets a specific limitation of traditional operator-based mutation testing: fixed mutation operators can only generate faults expressible by their predefined syntax transformations, while some real-world bugs require more context-specific changes. In a case study of 40 real-world bugs, LLMorpheus generated mutants that were syntactically identical to the buggy code in 10 cases and generated mutants that produced the same test failures as the real bug in 26 additional cases. These results suggest that LLM-based mutant generation can improve the expressiveness and realism of mutation analysis, though the problems of equivalent mutants and mutant execution cost remain relevant.

CHAPTER 4: RESEARCH QUESTIONS & METHODOLOGY

Chapter 1 introduced the goal of this thesis: evaluating whether an MCP-based, feedback-driven agent can generate effective JavaScript unit tests, and whether visible mutation feedback improves the quality of those tests. This chapter turns those goals into an experimental design. We reiterate on the main research questions and introduce additional validation questions. Finally, we define the experimental conditions, metrics, benchmark suite, and LLM settings used to answer them.

4.1 Research Questions

The main research questions of this thesis are:

> **RQ1: How effective is an MCP-based agentic test generation system for generating JavaScript unit tests?**
>
> **RQ2: How does incorporating mutation testing feedback through StrykerJS affect the quality of LLM-generated tests?**

RQ1 evaluates the complete system as a test generator. It is answered using the generated tests' pass rate, non-triviality rate, coverage, mutation score, mutant outcome breakdown, and token usage. RQ2 isolates the contribution of visible mutation feedback within the agentic system. It is answered by comparing an agent configuration where successful mutation feedback is hidden from the model with one where StrykerJS feedback is visible during generation.

For each of these main research questions, we provide two validation questions to guide our investigation:

> **VQ1.1: How does the correctness, coverage and mutation score of tests generated by the agentic approach compare to those generated by TestPilot?**
>
> **VQ1.2: How does the token and runtime cost of the agentic approach compare to TestPilot's light context test generation baseline?**

VQ2.1: How does mutation feedback targeted to the API item level affect project-wide mutation score?

VQ2.2: How does the residual mutation feedback phase affect mutation score relative to its additional token overhead?

These validation questions do not introduce separate research goals, but are used to systematically evaluate the existing ones. VQ1.1 and VQ1.2 guide the answer to RQ1 by evaluating how our agent measures against the JavaScript state-of-the-art, while VQ2.1 and VQ2.2 allow us to go in-depth for RQ2. If the answers to those validation questions significantly differ, it indicates that the answer to RQ2 depends on how and where StrykerJS' mutation feedback is integrated.

Variables and constants For RQ2, the only variable we want to evaluate is making per-item Stryker feedback visible to the model. What remains the same is: all other tooling, the benchmark suite, the LLM, and the external evaluation pipeline. Additionally, the separate residual feedback phase is naturally a variable because the final mutation score before the residual phase starts is always recorded. This means that the pre- and post-residual scores are available for all runs with a residual phase. For the mutation score achieved by an agent with access to item-level mutation feedback but no residual feedback phase, we use the term *pre-residual* mutation score.

4.2 Experimental Design

Our evaluation combines a comparison on the system level comparison and a controlled ablation study. The system-level comparison answers RQ1 by evaluating how effective our agentic approach is as a JavaScript unit test generator. The controlled ablation answers RQ2 by isolating whether making StrykerJS mutation feedback visible to the model improves the quality of generated tests. Together, these two parts let us evaluate both the complete system and the specific contribution of mutation feedback.

Comparison with TestPilot To evaluate the overall effectiveness of the agentic system for RQ1, we compare it with TestPilot, an existing LLM-based unit test generator for JavaScript (discussed earlier in 3.2.2). One problem with this comparison is that the tool-enabled agent approach is difficult to compare fairly against an LLM-based test generator without tool access. Our agent is free to call exploration tools when necessary and can, thanks to access to tools like `read_file` and the regex-enabled `search`, gather much more

information than is available in TestPilot's harness. However, we will not control for information budget, because the difference in context management is precisely what sets the two approaches apart. We can only compare the outcomes of each system, and do so according to VQ1.1 and VQ1.2. For test quality, we use VQ1.1 and measure test correctness, coverage, and mutation score. The latter is not available for TestPilot's published datasets, but they are complete enough to make retroactive mutation analysis relatively straightforward. To measure the effect of our approach on cost, we refer to VQ1.2 and track token expenditure and test generation time.

Mutation feedback ablation To answer RQ2, we follow controlled comparisons according to VQ2.1 and VQ2.2. The independent variables they introduce are whether successful StrykerJS mutation feedback is visible to the model, and at what point in the test generation process this is presented. In either case, project exploration tools, prompt preparation, benchmark suite, and LLM configuration are all kept constant.

To answer RQ2 by evaluating the variables presented by VQ2.1 and VQ2.2 requires us to evaluate three different agent modes:

- **Hidden mutation feedback:** the agent receives access to all tooling, but any mutation feedback is redacted.
- **Partial mutation feedback:** the agent receives model-visible StrykerJS feedback during test generation.
- **Full mutation feedback:** the partial feedback condition plus a residual mutation analysis phase.

To answer VQ2.1, we use the first and second condition. In the first condition, StrykerJS feedback is redacted from the tool response and detailed analysis is unavailable. Failed StrykerJS calls remain visible, because they can contain information about ordinary test failures that the model must repair before mutation testing can proceed. The second condition is exactly the same, but makes mutation feedback actually visible. This isolates the effect of the model seeing mutation analysis per API item. One limitation of keeping these 2 setups so similar is that we can do no meaningful comparison on the tokens used for both approaches: the hidden condition will waste tokens on executing Stryker calls and interpreting the redacted results, which means it is operating far from maximum efficiency. That is why we opt to only measure the token costs for the last condition.

Evaluating the third condition is straightforward. In any run with a residual feedback phase, we simply record the mutation scores and token usage before and after. This allows us to answer VQ2.2 without introducing an extra agent mode.

4.3 Metrics

We select metrics from the LLM-related unit testing work surveyed by Zhang et al. [71]. However, not all of the reported metrics are suitable for the evaluation of test quality. Examples from their report are 'correct rate', which is only applicable to generation of non-test code as it needs to satisfy a test suite. We select the following metrics:

- **Pass Rate:** The proportion of generated tests that execute without errors. The benchmark harness records the number of passing, failing, pending, and other tests after running the generated suite.
- **Non-triviality Rate (NT*):** The proportion of generated tests that contain at least one *non-trivial* assertion. Following TestPilot [55], an assertion is non-trivial if it depends on at least one function from the library under test, which we detect by tracing the assertion's backward program slice to an import of the package under test. The purpose of this metric is to separate tests that genuinely check library behaviour from tests that pass without asserting anything meaningful about it. This matters because code coverage alone can overstate test quality: a test may execute the code under test yet verify nothing about it, for example an assertion-free test that merely does not throw, or a vacuous check such as `assert.equal(true, true)`. Schaefer et al. introduce the metric precisely because automatically generated tests "often lack assertions, or only contain very generic assertions [...], or too many spurious assertions" [55]; a high non-triviality rate is therefore evidence that passing tests carry real oracles rather than merely inflating coverage. We reuse their definition and adapt the measurement script to fit our modern environment.
- **Code Coverage:** The percentage of the program under test that is actually executed by the generated tests. This is measured on the line, statement, branch, and function level.
- **Mutation Score:** The Stryker mutation score of the generated suite over valid mutants.
- **Mutant Outcome Breakdown:** Stryker provides the absolute counts of killed, surviving, no-coverage, timeout, runtime-error, compile-error, ignored, detected, and undetected mutants. These statistics complement the mutation score by revealing why mutants go undetected: surviving mutants indicate weak assertions over reachable code, while no-coverage mutants indicate unexplored code paths. We do not directly

use these metrics for systematic analysis, but record them to aid with explaining any anomalies encountered during the experiments.

- **Token Usage and Cost:** We record input tokens, cached input tokens, output tokens, reasoning tokens where available, total LLM requests, generation time, and estimated cost. Estimated cost is computed from these counts under the GPT-5.4 API rate in effect at the time of the experiments: $1.25 per million uncached input tokens, $0.125 per million cached input tokens, and $10.00 per million output tokens.

Together, these metrics capture runtime correctness, assertion relevance, code coverage, fault-detection effectiveness, and generation cost.

4.4 Benchmark

We evaluate the system on a shared benchmark suite used for the TestPilot comparison. We replicate the 25-package dataset from the original study, described in Section 3.2.2. This lets us compare the agent against TestPilot with the same LLMs, making a direct comparison possible. Appendix C summarises each library, including its domain, source size, mutant count, and any interesting features that might affect test generation or evaluation.

During the TestPilot replication with GPT-5.4, we marked five libraries in the suite as incompatible: `bluebird`, `rsvp`, `memfs`, `js-sdsl`, and `fs-extra` were skipped in all experiments for the reasons described in Section 6.1.2. The remaining 20 active libraries are checked out at the same commits listed in TestPilot's specification.

4.4.1 LLMs

We evaluate the system using OpenAI's GPT-5.4, a proprietary frontier model released in March 2026. OpenAI reported strong performance for GPT-5.4 on several agentic benchmarks at launch, including GDPval for professional knowledge work [48], SWE-Bench Pro for long-horizon software-engineering tasks [16], and Toolathlon for realistic multi-step tool use [36]. These results make GPT-5.4 a suitable model for evaluating the maximum effectiveness of our approach. Ideally, we would include a variety of models including open-source ones, but budget constraints prevent us from doing so.

4.4.2 Temperature

Temperature is a relevant factor in our comparison with TestPilot, but it is not fully controlled. The original TestPilot evaluation used low-temperature decoding: for its main

`gpt-3.5-turbo` experiments, it sampled completions at temperature 0, while its StarCoder experiment used temperature 0.01 because that model did not support temperature 0 [55]. Our GPT-5.4 TestPilot replication follows this low-temperature setup and runs at temperature 0. By contrast, the main agent runs in this thesis were executed at temperature 1.

Temperature 1 is a relatively high setting for an evaluation experiment. Higher temperatures allow more variation in the model's completions, which can increase exploration and produce a wider range of possible test cases. However, they also make runs less deterministic and can increase the risk of unsupported API assumptions or invalid assertions. This setting was initially used as the default generation configuration for the agentic loop, but in retrospect it introduces an avoidable difference between the TestPilot and agent runs.

The TestPilot comparison therefore cannot be interpreted as a comparison between only two generation architectures. It also compares a low-temperature TestPilot run with a higher-temperature agent run. The direction of this effect is not obvious: a higher temperature may help the agent discover more edge cases or alternative test designs, but it may also reduce reliability. To estimate the sensitivity of our conclusions to this setting, we include an additional temperature 0 run of the full mutation-enabled agent condition. This run is not part of the main controlled mutation-feedback ablation, but serves as a robustness check for the TestPilot comparison.

This limitation does not affect the mutation-feedback ablation in the same way. The hidden-feedback and full-feedback agent conditions are run with the same agent configuration and temperature, so RQ2 still evaluates the effect of mutation-feedback visibility. Temperature is primarily a threat to validity for the validation questions that compare the agent against TestPilot.

4.5 Runs and Replications

This section details the concrete set of experiments we run to evaluate the conditions (Section 4.2) over the benchmark suite (Section 4.4).

For each condition, we use 2 temperature-1 runs to mitigate the effect of the run-to-run variation. This means we need 4 temperature 1 runs in total: 2 executing the hidden-feedback condition, and 2 executing both mutation-enabled conditions simultenously. As explained previously, we can evaluate both conditions in the same run because the pre-residual score is recorded even in runs with a residual phase.

All four temperature-1 runs use the same agent version and the same flags, and differ only in whether successful mutation feedback is visible to the model. We additionally execute the mutation-enabled conditions once at temperature 0 as a robustness check for the TestPilot

comparison described in Section 4.4.2. This brings the evaluation to 5 agent runs in total, as shown in Table 4.1.

Table 4.1: Agent runs included in the evaluation and the conditions they evaluate.

Run set	Mutation feedback	Temp.	#Runs	Conditions
Hidden	redacted	1.0	2	1 (hidden)
Full	visible	1.0	2	2 (partial), 3 (full)
Full	visible	0.0	1	robustnesss check
Total			**5**	

CHAPTER 5: ARCHITECTURE & IMPLEMENTATION

Chapter 4 defined the experimental design used to evaluate the MCP-based agentic test generation approach. This chapter describes the architecture and implementation of the agent and research harness used to execute that design. The system is built to generate JavaScript unit tests under controlled experimental conditions and vary the visibility of StrykerJS mutation feedback while keeping the remaining tool support constant.

The chapter first gives a high-level overview of the system components and their data flow. It then explains the main design decisions shaping the agent, before separately describing the implementation details of the agentic generation loop, the MCP tool integrations, robustness mechanisms, and report formats.

5.1 System Overview

Figure 5.1 shows an overview of the test generation workflow implemented by the research harness. The process starts when the benchmark harness is invoked for a target library. The harness prepares the library, starts the required MCP servers, and performs API extraction. This step identifies the exported functions and methods that will become "todo items" for the agent.

For each item, the harness constructs a prompt containing the API access path, the expected test file, a bounded source snippet, and the rules and budgets that apply to the item. The agent then enters a feedback loop in which it writes or edits tests, validates them, and requests mutation feedback through the Stryker MCP server. To save computation time, Stryker is configured to only mutate the code in the source range associated with the active API item. The returned mutation feedback is used by the agent to strengthen the tests before closing the item.

After the item is completed or dropped (Section 5.3.5 explains how this happens), the harness validates the result and records a structured JSON summary emitted by the model. When all items have been completed, the harness checks the full generated suite for cross-test failures, possibly starting a repair phase before moving on. Once every item has been processed, a whole-project Stryker run produces the mutation score.

The optional residual feedback phase starts from this aggregate mutation report, but is omitted from Figure 5.1 to keep the main workflow readable. Its motivation is discussed in

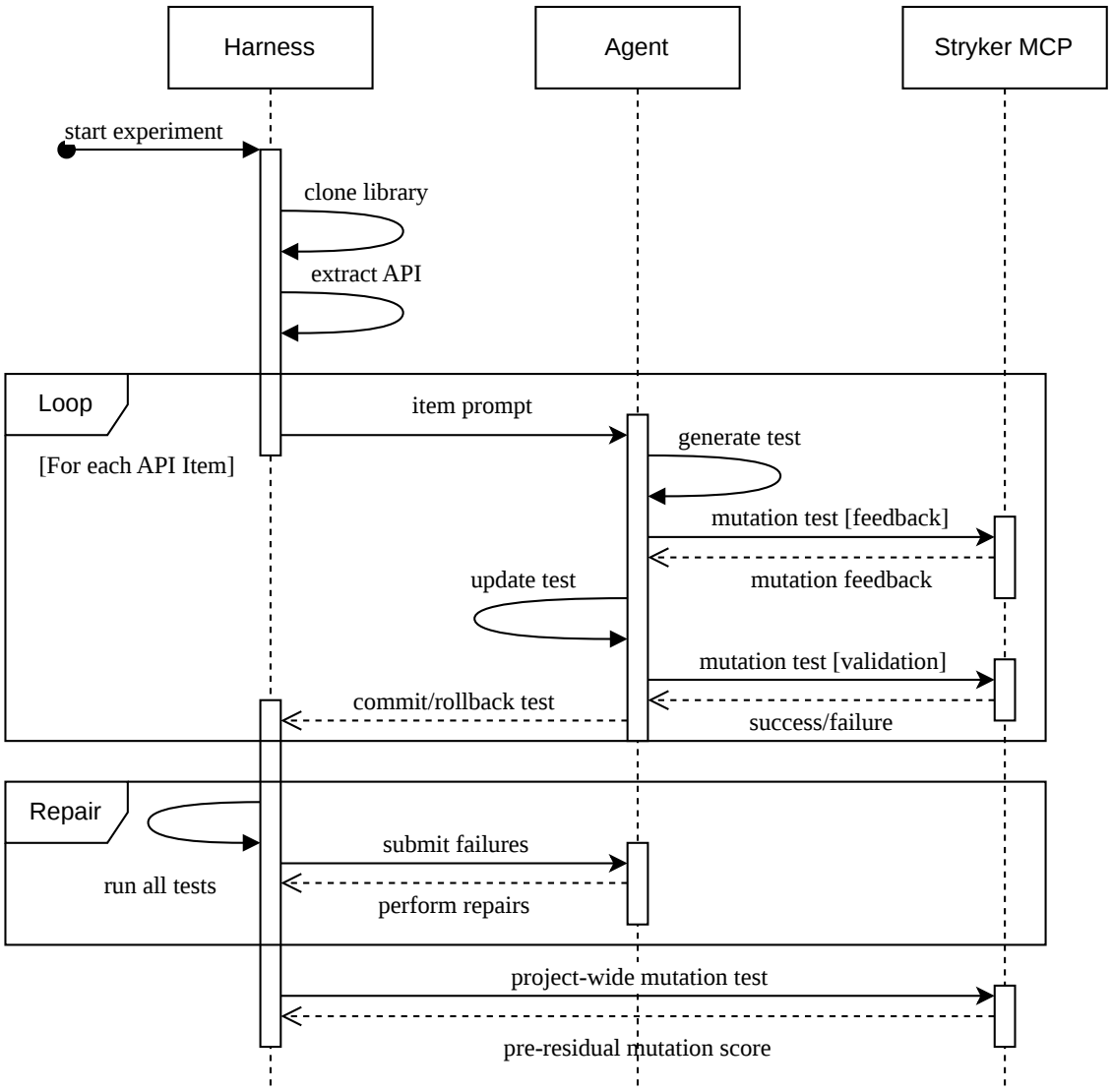

Figure 5.1: Sequence-style overview of the agentic test generation workflow, excluding the residual feedback phase. The non-Stryker MCP tools were omitted for readability. Together with other lower-level details, they are discussed in later sections.

Section 5.2.6, and its implementation is described in Section 5.3.6.

Throughout the process, the harness records a large variety of data, including generated tests, LLM transcripts, verdicts, token usage, test results, coverage, and mutation outcomes.

5.1.1 Repository Structure

The system consists of five TypeScript packages in a single repository. Table 5.1 shows each package's high-level responsibility.

Package	Role
agent	Command-line interface and test generation loop. Connects a language model to MCP tool servers through an MCP client, with provider-agnostic model integration through LangChain.
mcp-server	Exposes MCP tools: StrykerJS, test runner, source snippet reader, and public-API explorer.
benchmark	Executes the agent on containerized benchmark libraries, validates the generated tests, and compiles the resulting reports.
report-schema	Shared schemas used by the agent and benchmark for reports, transcripts, and coverage data.
example-app	Small local target for testing & developing new features.

Table 5.1: Packages in the repository and their roles.

5.2 Design Rationale

This section explains the reasoning behind the main design decisions in the harness. Several of these decisions were motivated by failures observed in an earlier pilot run of a simpler agent architecture. Table 5.2 summarizes these observations and links them to the corresponding design choices. The subsections following the table follow the same order: per-item generation, on-demand code search, and scoped mutation feedback. The final subsections discuss two later refinements: bounded item budgets and residual feedback.

5.2.1 Pilot Observations

The final architecture was informed by decisions made on an earlier run of a simpler agent architecture. This agent had access to the same LLM configuration and mutation testing tools, but no explicit API exploration phase or code search. The only way to perform project exploration was exposed by the filesystem MCP server in the form of coarse-grained tools like

`read_file`, `read_multiple_files` and `search_files` (only searches for filenames). This proved to be too coarse to scale across bigger libraries, and led to a number of changes. A summary of the problems we found is available in Table 5.2, and the supporting data is in Appendix A.

Table 5.2: Problems in the pilot run and their correlation to the final architecture. Detailed per-library numbers are available in Appendix A.

Observation	**Evidence**	**Design choice**
Keeping full conversation history can become expensive	A repository-wide search result carried throughout the entire conversation inflated the input token count of a small project.	Per-item conversations.
Project exploration with only filesystem tools is inefficient and unreliable	Multiple generated test suites failed due to incorrect imports, setup, or usage assumptions.	Add code search through a ripgrep MCP server.
The agent initiated almost exclusively project-wide mutation tests, which can make iteration unnecessarily expensive	A full Stryker run timed out before feedback reached the model on a package.	Programmatically restrict per-item Stryker calls to file/line scopes of the active item.

The following subsections go into more detail on Table 5.2. Each subsection explains how one pilot observation led to a design decision in the final harness. The final two subsections describe design choices that were introduced later: bounded item budgets and residual feedback.

5.2.2 Per-Item Generation

The first observation in Table 5.2 concerns the cost of carrying full conversation history through an entire library run. This motivated the per-item generation design: instead of maintaining one long conversation per package, the harness starts a separate agent run for each public API item. This avoids unbounded growth from raw tool output bloating the context window, and prevents possibly irrelevant information from one API item influence the test generation of another. There is a clear example of this in the pilot run: on uneval, a single file read early in the generation loop returned the contents of the entire project (≈245k

tokens). This was re-sent in the 10 following model calls, and accounted for roughly 2.45M of the 2.57M input tokens (Table A.3).

To allow a small amount of information to cross item boundaries, the agent returns a structured verdict at the end of each item. This verdict can include short item-specific and library-wide notes. The implementation of this mechanism is described in Section 5.3.5.

5.2.3 On-Demand Code Search

The second observation in Table 5.2 concerns project exploration. The agent struggled with filesystem-only exploration: Of the 8 libraries with no passing tests, 6 failed simply because the agent could not locate correct imports, setup code, or usage examples through directory listings and file reads. These libraries are glob, graceful-fs, pull-stream, jsonfile, geo-point, and gitlab-js (Table A.1).

Given the time constraints of this thesis, an existing code-search MCP server backed by `ripgrep`, a fast command-line search tool, provided a simple implementation path. It addressed the immediate problem observed in early runs: locating relevant definitions, usage examples, and import patterns without loading large parts of the project into the model context.

5.2.4 Scoped Mutation Feedback

The third observation in Table 5.2 concerns the cost in runtime of unconstrained mutation testing. In early runs, the agent often requested project-wide Stryker runs, which could time out before useful feedback reached the model. This motivated scoped mutation feedback.

On `countries-and-timezones`, which is a small library dense in mutants (78 lines / 2,663 mutants), both of the agent's `strykerMutationTest` calls exceeded the MCP tool timeout before the agent finished at 8.1% mutation score (Table A.1). On omnitool, the largest library in the suite (8,904 mutants), all three Stryker attempts also failed to return a result: one dry-run failure followed by two MCP timeouts. These runs were effectively wasted, but provided quick feedback that leaving the scope of mutation tests to the agent might not be a reliable solution. First, the MCP timeouts were doubled, to support project-wide mutation testing where it is necessary. Then, we opted to leverage the per-item conversation design choice by pre-calculating the source range of the active API item. This forces efficient, targeted mutation testing during iteration, while project-wide mutation testing is now only available in repair phases and evaluation of the final result.

For example, the mutation feedback calls for the API element `getAllCountries` extracted from `countries-and-timezones` are constrained to its specific source range:

```
strykerMutationTest({
  mode: "files",
  files: [{
    path: "dist/index.js",
    range: {
      start: { line: 4052, column: 3 },
      end: { line: 4057, column: 4 }
    }
  }]
})
```

This saves the Stryker Server from mutating more than 4000 lines of code that are not relevant to the tests being generated.

5.2.5 Bounded Item Budgets

Bounded item budgets are a consequence of the same failure modes summarized in Table 5.2. Per-item generation limits context growth, but without explicit tool-call budgets, a difficult item could still consume disproportionate time or tokens.

We opt for two tool-call budgets: one for mutation testing and one for all other tool calls (mostly project exploration). The reason for this is that we are trying to keep 2 factors within reasonable limits: token use and run time. Token use is mainly controlled by project exploration and test (re)write, while the mutation analysis easily dominates the runtime of the agent. We therefore separate these limits into one budget which bounds total tool use, and a budget which bounds successful mutation feedback calls. However, failed Stryker calls do not count against the latter, because failed dry-runs require much less computation time than a full mutation run and provide useful feedback for repairing tests.

5.2.6 Residual Feedback

The residual feedback phase addresses a new limitation introduced by the scoped-feedback design (Section 5.2.4): some surviving mutants are reachable through public API calls but are not directly in the source code range computed for an API item. For example, public export can be a small wrapper around an internal helper, or some bundled or compiled packages can have some behaviour defined outside the computed source ranges. The generated tests can still reach that code through calling the API item, but the mutation tests might not be able target it.

This gap appeared in development runs after the introduction of the API extraction. On the `plural` package, several runs of the scoped loop produced low aggregate mutation scores, despite completing the discovered API items. The reports shows why this failure mode is plausible: API item `plural.addRule` had a calculated source range of lines 9-12, while many surviving mutants were clustered in the internal `addRule` implementation around lines 16-70. Those mutants were reachable through the public API, but they were not owned by any active API item's mutation-feedback range.

We considered some alternatives but ultimately deemed them less suitable: Allowing the model to decide its own mutation testing scopes could in theory be the perfect solution. However, the results would be highly dependent on the capabilities of the LLMs to correctly derive these scopes and formulate them as Stryker inputs, while they previously lazily opted for full-size runs for almost every tool call (Section 5.2.4). Additionally, we would lose the guarantee that a test emitted for an item only concerns that item's source range. This would undermine the repair phase. A test attributed to item X but failing on item Y's code would send X back to fix a problem that does not lie in its source range. Alternatively, showing the full aggregate Stryker result to the LLM as one large feedback object would bypass the work-queue structure and reintroduce the same cost and context problems that scoped feedback was designed to avoid.

The final design therefore keeps scoped mutation feedback as the default per-item mechanism, but adds a separate residual phase after the first aggregate mutation run. The purpose of this phase is to compensate for mutants that are reachable through the public API while falling outside the source ranges assigned to regular API items. Section 5.3.6 describes how these residual regions are selected and converted into additional work items.

5.3 Agentic Test Generation Loop Implementation

Figure 5.1 introduced the main control flow of the generation loop. This section describes the implementation details that were abstracted away: how API items are discovered, how prompts are constructed, how test execution and mutation feedback are gated, and how completed items are attributed to generated files.

The harness decides what is being tested, while the agent decides how to test it: The harness constructs the prompts for each API item, alongside source ranges, expected filenames, tool budgets, and validation rules. Within these boundaries, the LLM can inspect the project, write and revise tests, and request mutation feedback.

5.3.1 API Exploration

Before the first model call, the harness invokes `exploreAPI` once on the package under test. This function is inspired by TestPilot's API exploration phase (see Section 3.2.2) and returns a list of exported functions and methods. An example entry in the list could be `simple-statistics.mean(x: number[]): number`. The results of exploreAPI are also stored inside the Stryker MCP server. Details like signature, recovered body, and source location are stored for later use by getFunctionBody.

The explorer uses two strategies: When declaration files or TypeScript sources are available, it uses the TypeScript compiler API to recover exported symbols and typed signatures without executing the package. If that strategy produces no functions, we fall back to a sandboxed child process that loads the package and walks the resulting exports object (this is the exact strategy implemented by TestPilot). The resulting access paths become the initial todo list.

5.3.2 Prompt Construction

Each API item in the queue is handled by a fresh `agent.run()` call. Its prompt is assembled by the harness' item prompt renderer, and consists of a mostly stable instruction prefix (to enable token caching) with a dynamic item suffix. The dynamic part contains the current work-queue summary, the library notes collected so far, the active signature and access path, the source snippet returned by getFunctionBody, the name of the test file the model should write, the exact runTests pattern it should use, and the current tool budgets.

If the function body for the item is too large, the prompt includes a truncated snippet and directs the model to use `read_file_range` or ripgrep if it needs detailed information. This keeps the starting prompt small, but leaves space for the agent to do what agents are good at: fetching more context when it is necessary.

Appendix B contains an abbreviated example of a rendered per-item prompt. The full prompt is longer because it includes several extra harness rules for imports, filesystem safety, mutation feedback, and formatting.

5.3.3 Test Validation

During the test generation phase, the agent cannot start a mutation test without a preceding successful `runTests` call. After every write or edit, the model must run the targeted test pattern for the active item. The prompt states this pattern explicitly, and the harness rejects `runTest` calls that cover more files than the predefined patterns. These patterns are derived

from the access path and match the item's expected test filename plus optional extra files with the same prefix.

Furthermore, if the same exact test failure appears twice in a row, the item is stopped immediately. This prevents the model from spending the entire budget retrying the same broken assertion.

5.3.4 Iterative Mutation Refinement

Once the generated tests pass, the model can request mutation feedback. As discussed in Section 5.4.2, successful Stryker results are converted to compact plaintext. During per-item generation, the only permitted mutation call is `mode: 'files'` on the active item's file and line range. Calls to `mode: 'all'`, `mode: 'survivors'`, or `mode: 'mutants'` are MCP-server capabilities, but the harness' tool use policy rejects them during this phase.

Each successful mutation run does count against the item's mutation budget. When that budget is hit, the item is forcibly closed, but the tests are recorded (if they are passing).

To support the controlled ablation for RQ2, the harness can hide successful mutation feedback from the model. In that case, the same Stryker calls are still fully executed so their outputs can be recorded in the reports, but receive mutation feedback is redacted for the model. Likewise, `strykerMutantDetails` is not available, but mutation testing errors remain visible so that test failures can still be repaired.

5.3.5 Verdicts and Test Attribution

An item ends when the model returns a small JSON summary in its final assistant message. The verdict has a completed/skipped status, an optional skip reason, optional notes for the item, and an optional line to append to the library-wide notes.

When the verdict arrives, the harness checks that the expected file or a valid file with the correct suffix exists. If not, the item is force-closed as skipped. This guarantees that repair phases can correctly trace back failing test files back to the item that 'owns' them.

5.3.6 Residual Mutation Feedback

The motivation for this phase was discussed in Section 5.2.6; this subsection describes its implementation. When residual feedback is enabled, the harness performs an aggregate Stryker run after the regular API-item loop and repair phase. This run is stored as `preResidualAggregatePass`. The harness filters out clusters that already belong to source ranges processed by regular API items.

The remaining clusters are sorted by survivor count and source location. Large clusters are split into bounded source regions before the harness appends up to `-residual-mutation-max-items` synthetic items to the work queue. The default cap is five, and the cap applies after splitting. Each synthetic item receives an identifier such as `__residual__.index_js.addRule.L16-49.part1of3`, together with a source snippet and file range.

Residual items are processed by the same loop as regular API items, but their prompt presentation is different. Instead of asking the agent to test a specific exported function or method, the prompt asks it to exercise a source region through the public library interface. After all residual items have completed or been skipped, the harness performs a final aggregate Stryker run. This final pass provides the reported benchmark mutation score, while the pre-residual pass is retained to measure the effect and cost of the residual phase separately.

5.4 Tool Integrations

The previous section described how the generation loop uses external tools. This section describes the MCP tool integrations that provide those capabilities. All tools are made available through servers adhering to the Model Context Protocol (see Section 2.2.1). Custom tools are exposed through our own Stryker MCP server, while filesystem access and code search are provided through separate MCP servers.

In the remainder of this section, we discuss the test-execution tool, the StrykerJS mutation-testing interface, filesystem access, and ripgrep-based code search.

5.4.1 Test Execution

The Stryker MCP server exposes `runTests` as a lightweight test-execution tool. It accepts a glob pattern, runs the matching generated test files through Mocha, and returns a compact pass/fail summary. When a test fails, the response includes the failing test names and short error snippets that can be fed back to the model.

This tool is used by a validation step in the agentic test generation loop, described in Section 5.3.3. Keeping plain test execution separate from mutation testing allows the harness to check generated tests before invoking the more expensive StrykerJS mutation analysis.

5.4.2 Mutation Testing with Stryker

Stryker MCP interacts with Stryker Server, an execution mode for StrykerJS that adheres to the Mutation Server Protocol [60]. It supports JSON-RPC methods for configuration,

mutant discovery, and mutation testing. In order to save tokens from the agent's context, we expose only a custom interface to the mutation testing tool, and a tool for retrieving extra information about undetected mutants. We briefly discuss these two tools below.

strykerMutationTest

The Stryker MCP server exposes an interface for mutation testing through its custom `strykerMutationTest` tool. It supports 4 custom execution modes: 'all', 'survivors', 'files' and 'mutants'. This means the agent can choose to perform mutation testing on all mutants in the project, only the survivors of a previous run, only the mutants in specific files (and line ranges), or specific mutants from a previous run. In the current agent, the harness applies a stricter policy on top: per-item calls must use mode: 'files' with the active item's file and line range, and whole-project 'all' calls are reserved for the final aggregate pass (Section 5.2.4, Section 5.6.1).

The output has been minimized from Stryker Server's verbose mutation-test result to a custom plaintext grouped overview. By default, `strykerMutationTest` clusters undetected mutants, meaning Survived and NoCoverage mutants, by file, nearby line range, and best-effort enclosing symbol. The output starts with a run summary containing the run ID, mutation score, total valid mutants, and total undetected mutants, then emits one line per shown cluster with the cluster's undetected count, coverage summary, mutant IDs, and mutators. The tool also outputs one example mutation for each cluster up to a configured budget (default 10), using a diff-style original/mutated view. Listing 5.1 shows an example output for a successful tool call with 3 remaining undetected mutants.

```
RunId: 0  Score: 93.02% (95.24% covered)  Mutants: 43  Undetected: 3

Clusters (1, covering 3 undetected):
  src/isPrime.js
    L2 @ isPrime  x3  cov=partial  ids 18,19,21  ConditionalExpression*2,
       EqualityOperator

Representative diffs (1 of 1, one per cluster):
  src/isPrime.js:2:6 id=18 ConditionalExpression
    - if (n <= 1) return false;
    + if (false) return false;

Drill down:
  - strykerMutantDetails(runId, refs=[{filePath, id}, ...]) for specific
     mutants
  - strykerMutationTest({mode:'files', files:[{path,
     range:{start,end}}]}) to re-score one cluster
```

Listing 5.1: Example mutation test output (grouped format) with 3 undetected mutants

When mutation testing fails, the Mutation Server Protocol provides no standard error format. We therefore capture the final output lines emitted by the Stryker process and parse them into a minimal error message. This is important because test failures are a natural part of test generation. If the agent produces syntactically invalid tests or incorrect assertions, Stryker's initial dry-run aborts before mutation testing starts. This avoids spending mutation-testing time on a broken test suite; the corresponding repair phase is described in Section 5.5.4. These dry-run errors are compiled into a list of failing tests with concise descriptions of their failures, as shown in Listing 5.2.

```
One or more tests failed in the initial test run:
isPrime small primes
    Error: expect(received).toBe(expected) // Object.is equality
Expected: false
Received: true
```

Listing 5.2: Example mutation test output with assertion errors

strykerMutantDetails

strykerMutantDetails is an extra custom tool for obtaining more detailed information for a specific surviving mutant. It takes the original mutation test runID and a mutant ID and

outputs an expanded source snippet alongside precise information on the location, mutation, and test coverage of that mutant. Listing 5.3 shows a call for details on the ConditionalExpression survivor from Listing 5.1 and the structured response the tool returns.

```
{ // request
  "runId": 0,
  "refs": [{ "filePath": "src/isPrime.js", "id": "18" }]
}

{ // response
  "runId": 0,
  "mutants": [
    {
      "filePath": "src/isPrime.js",
      "id": "18",
      "found": true,
      "mutant": {
        "id": "18",
        "mutatorName": "ConditionalExpression",
        "status": "Survived",
        "location": {
          "start": { "line": 2, "column": 6 },
          "end":   { "line": 2, "column": 12 }
        },
        "replacement": "false",
        "coveredBy": ["1", "2"]
      },
      "sourceSnippet": {
        "startLine": 1,
        "endLine": 5,
        "text": "function isPrime(n) {\n    if (n <= 1) return
           false;\n    for (let i = 2; i * i <= n; i++) {\n
           if (n % i === 0) return false;\n    }"
      }
    }
  ]
}
```

Listing 5.3: Example strykerMutantDetails call and response

5.4.3 Filesystem Access

In order to read existing source code and write new test files, we provide the agent with access to the MCP filesystem server '@modelcontextprotocol/server-filesystem' maintained by Anthropic [43]. The agent receives no shell access, and the filesystem server is configured with restricted roots covering the project tree and the dedicated test-output directory, so the agent cannot read or write outside the project..

We disallow some tools from the server to reduce context bloat and to avoid redundancy with custom tooling. For example, tools that return too much information (`directory_tree`, `list_directory_with_sizes`) can flood the prompt with content that does not help test generation. The one-level `list_directory` tool should be sufficient for basic exploration, and code search (Section 5.4.4) covers the case where the agent needs deeper navigation. Likewise, some file operations (`move_file`, `get_file_info`, `read_media_file`) are not required by the agent. Those have been removed to reduce context bloat.

The agent therefore sees only the following files from the MCP filesystem server:

- `read_text_file`, `read_multiple_files`: reading source files and previously written tests
- `list_directory`, `list_allowed_directories`, `search_files`: project exploration
- `create_directory`, `write_file`, `edit_file`: creating and refining test files

One problem with the off-the-shelf filesystem server is that it only allows reading full text files. With the narrow per-item scope in our generation loop, that can often be overkill. For that reason, we provide a custom `read_file_range` tool that takes a file path together with a start and end line. The tool returns just the requested slice plus the total line count. This supports use cases like inspecting the body of a function surrounding a surviving mutant or the source code around a snippet matched by a `ripgrep` query.

Our tool is also limited to the project directory and rejects paths outside it, so it does not give the agent more access than the filesystem server already permits.

5.4.4 On-Demand Code Search

We expose the code search capability (motivated in Section 5.2.3) through an off-the-shelf MCP server: `mcp-ripgrep` [12], which wraps the `ripgrep` command-line tool. The agent can use it to make a regex query inside the project, optionally filtered by a file range. The tool returns all matching lines plus a small amount of surrounding context. This means that

the token cost no longer scales with the size of the file in which the answer lives, but with the size of answer itself.

The tool can be used to approximate relationships between code, such as which function calls which other function, which export resolves to which definition, or which tests exercise a particular path. However, this depends on the LLM constructing useful regular-expression queries.

5.5 Robustness and Hygiene Mechanisms

The harness also has several safeguards that help the agent stay on track, aimed at increasing the probability that the generated test suite is valid after a long-running agent session.

5.5.1 Budget Enforcement

The budgets motivated in Section 5.2.5 are enforced by a callback around MCP tool calls. It gives us enough context to determine which tool is being called, making the decision to decrement the correct budget a very straightforward implementation: if the tool budget is exhausted, any tool call aborts the item. The next Stryker attempt after the successful Stryker call budget is exhausted is blocked before it reaches the MCP server. The actual values of the budgets are configurable, however they are not part of the research question because we do not have the budget to investigate multiple choices.

5.5.2 Transactional Test Writes

At the start of each item, the harness records the contents and filenames of every file in the generated-test directory. If the item completes and passes harness validation, the snapshot is committed and the agent's tests are kept. If the item contains failing tests after close, exceeds its budget, or the model returns an invalid verdict, the transaction rolls back to the snapshot. This prevents a failing item from leaving broken tests that would poison later items or the final run.

5.5.3 Suite Repair Phases

Tests that pass in isolation can still fail when the full generated suite runs together. This is a problem that some of TestPilot's test suites face, as discussed in Section 6.1.2. After each completed item, the harness checks the full generated test suite with `runTests`. If the check fails, the failing files are mapped back to their owning items through the filename convention

and those items are marked for repair. A final repair pass runs after all pending items have been processed.

Repair prompts consist of the failing test information and the relevant generated test files. They also forbid mutation testing, which is made unavailable by the harness for the duration of the repair phase. The goal of repair is to get the full generated suite passing again before doing the full Stryker evaluation, not further iteration on mutation feedback. The repair loop stops when the suite passes, when the exact same failure repeats, or after a hard cap of repair iterations.

5.5.4 Stryker Dry-Run Repair

As discussed in Section 2.3.3, Stryker starts every mutation test with an initial dry run, whose purpose is to abort mutation testing if the unmutated test suite already fails. Although this dry run also simply executes the generated tests, it is not the same process as the harness's direct Mocha check. The harness calls Mocha through `runTests`, while Stryker runs the suite through its own test-runner setup and Stryker configuration. Small differences in module loading, global state, test ordering, or timeouts could therefore make a suite pass under direct Mocha execution but fail during Stryker's dry run. In our experiments this occurred on `q`, where a generated test depended on the value of `this` inside a bare function call. That value differed between direct Mocha execution and Stryker's test-runner environment.

The harness therefore implements a Stryker dry-run repair phase. It is triggered only after normal repair has passed and the full Stryker pass fails its dry-run.

In this phase, the model receives the Stryker dry-run failure text and a repair prompt, but it still cannot call `strykerMutationTest`. After each repair attempt, the harness itself retries the aggregate mutation pass. The first successful retry is reused as the aggregate score input. This phase is recorded separately in the report, because it does not provide mutation feedback.

5.6 Evaluation and Reports

5.6.1 Aggregate Mutation Pass

After each item has finished generation and the repair phases are done, the harness performs a whole-project Stryker call with `mode: 'all'`. The result of this run is the mutation score reported by the benchmark. The aggregate pass uses a longer timeout than interactive item feedback, because full-library mutation testing can take a long time.

If there are still failing tests after the repair phase, this evaluation is skipped and the reason is recorded. When residual feedback is enabled, the first aggregate pass is stored as `preResidualAggregatePass`, after the residual feedback phase is started and a last full Stryker pass becomes responsible for the final score.

5.6.2 Report Artifacts

Each library run produces a `report.json` file containing configuration, token usage, per-item summaries, repair summaries, and coverage and mutation metrics. The report schema is shared through a separate `report-schema` package so that the agent and benchmark agree on the same contract.

The run also records the conversation with the LLM in a separate directory. The raw transcript with conversation and tool calls is recorded alongside `verdicts.jsonl` and various debug telemetry. For example, `loop-events.jsonl` records the harness-side events outside the per-item generation loop, like item selection and repair phases.

One helpful feature is that reports are incrementally written and saved throughout the run. The harness updates the report after API exploration, item closure, repair milestones, residual planning, and aggregate evaluation. This is especially important for long benchmark runs. If library terminates early or times out, the partial report still usually explains when the failure happened and what might have caused it.

CHAPTER 6: EXPERIMENTAL RESULTS

This chapter presents the experimental results for the study defined in Chapter 4. First, Section 6.1 establishes the results achieved by replicating the TestPilot experiments. It reports on the results of retroactive mutation analysis on the original TestPilot outputs, explains replication issues, and evaluates our GPT-5.4 replication. Section 6.2 then presents the results of the complete agentic system.

Section 6.3 puts the results of both approaches side by side, a comparison that will serve as evidence when answering RQ1. Because the agent runs used temperature 1 while the TestPilot baseline used temperature 0, Section 6.4 checks whether the results are affected by that temperature difference using an additional temperature 0 agent run. Finally, Section 6.5 provides data for RQ2 by reporting how mutation feedback affects generated tests for the hidden, partial, and full feedback conditions.

6.1 TestPilot Replication

First, we reproduce the results of the existing JavaScript test generator, previously discussed in Section 3.2.2, TestPilot [55]. The authors provide data for three different LLMs, and see the best results with OpenAI's GPT-3.5 Turbo. Since then, newer LLMs have shown stronger coding capabilities on real-world software-engineering benchmarks and benchmarks for LLM tool use and API calling [33], [50]. Our agentic approach is only suited for models with tool calling capabilities, so we use a GPT-5.4, a modern member of OpenAI's GPT model family (of which TestPilot's top performer GPT-3.5 Turbo is also a member).

6.1.1 Retroactive Mutation Analysis

Mutation analysis is a central part of our agentic approach, and can be seen as a proxy for test quality [1]. Analyzing the mutation scores achieved by TestPilot, which does not provide mutation feedback, can help answer RQ1. By retroactively performing mutation analysis on original test suites generated by TestPilot's authors, we establish a baseline of older models' performance. We clone the libraries to a docker container, install Stryker, and collect the mutation testing metrics. However, mutation analysis is not possible for the datasets of all libraries. The next sections report on why several libraries were dropped entirely, or required the exclusion of problematic tests.

6.1.2 Replication and Validation Issues

Some libraries could not be evaluated with the modern Node 22/Stryker toolchain, while other libraries produced tests that passed TestPilot's original per-test validation but failed when executed as a complete suite during Stryker's dry run. We distinguish three categories: incompatible libraries, independence failures, and environment overfitting.

Incompatible libraries The following libraries were excluded from the mutation-score comparison because the pinned package version or generated test suite could not be evaluated reliably in the replication environment:

`bluebird` is a library implementing promises, later succeeded by JavaScript-native Promises. Its API allows enabling 'long stack traces' by calling `bluebird.longStackTraces()`. This debugging mode records additional information when promises are created so that errors in asynchronous promise chains can be traced back through earlier asynchronous calls. Because that information must be captured from the start, Bluebird requires long stack traces to be enabled before any Bluebird promise has been created. If an earlier test has already created a promise, bluebird fails: `Error: cannot enable long stack traces after promises have been created`. In practice, this happens during every Stryker dry run on the TestPilot `bluebird` dataset, preventing the generation of valid mutation scores.

`rsvp` is another Promises implementation. It uses an old Ember CLI based build stack that does not install under our Node 22 toolchain.

`memfs` is an in-memory implementation of Node's `fs` API. The pinned commit uses an old TypeScript/Jest setup, including a `typescript@4.9.5` peer dependency that does not work with our Node/pnpm/TypeScript toolchain.

`js-sdsl` is a JavaScript/TypeScript library inspired by the C++ Standard Template Library. It is marked incompatible because the build writes CommonJS output while compiling with ES2015 module syntax. As a result, importing the package can fail on modern Node versions before any generated tests or mutation analysis can run.

Environment overfitting A problem occurs when tests depend more on the TestPilot environment than we can replicate. Due to TestPilot's iterative nature (one of its prompt refinement strategies includes allowing the LLM to rewrite a test with information on why it failed previously), it is possible for the test cases to overfit to the environment.

fs-extra is a library with clear examples of this problem, and was dropped from the benchmark because of it. Consider the generated test case in listing 6.1:

```
it('test case', function(done) {
    fs_extra.readdir('test', function(err, files) {
        assert.equal(files.length, 2);
        done();
    });
})
```

Listing 6.1: Expecting 2 files in test/

This test assumes the existence of a 'test' directory and asserts the presence of 2 files, testing the readdir function exposed by fs-extra. Coincidentally, there happens to be a test directory in TestPilot's execution environment: the one containing unit tests for TestPilot itself! The test opens this directory and then fails because there are actually 10 unit tests and 1 subfolder present, resulting in 11 items. Next, this prompt is refined with TestPilot's 'RetryWithError' strategy. This prompt is listed in listing 6.2.

```
it('test fs-extra.readdir', function(done) {
    fs_extra.readdir('test', function(err, files) {
        assert.equal(files.length, 2);
        done();
    });
})
// the test above fails with the following error:
//   11 == 2
// fixed test:
it('test fs-extra.readdir', function(done) {
});
```

Listing 6.2: Refining the prompt with an error message

The LLM learns that there are 11 files present in the test folder and updates the test case to reflect that (as shown in listing 6.3). This test passes in the TestPilot environment and is included in the coverage statistics. However, in our replication the tests are re-executed in a Stryker sandbox where there is no test folder present. Instead, the test now counts the members of fs-extra's own test folder, finds 1 item, and fails.

```
it('test case', function(done) {
    fs_extra.readdir('test', function(err, files) {
        assert.equal(files.length, 11);
        done();
    });
});
```

Listing 6.3: Passing test generated with the refined prompt

Independence Failures TestPilot spawns one validation process per generated test using the Mocha test runner for JavaScript. If the test succeeds, Mocha generates a coverage report. When all tests have been generated and validated, these reports are merged to obtain comprehensive coverage analysis of the entire library under test. In contrast, Stryker always performs its dry run before mutation testing can start. The dry run executes all tests specified through the configuration file as one test suite (as mentioned in Section 2.3.3).

Some test suites generated by TestPilot fail this dry run unexpectedly, despite all the individual tests passing in their isolated Mocha runs. We mitigate this by discarding tests that fail the initial test run, instead of excluding the libraries completely. In total, we discard 1.4% of tests for GPT-5.4-500 and 1.8% for GPT-5.4-1000. A simple example of this problem occurs in almost every test suite generated for the `plural` library, which provides methods for modifying the library's global state.

`plural` is a small library for pluralizing words. Amongst others, it exposes the functions `plural(word, num)` and `addRule(match, result)`. The plural function will return the plural or singular version of the provided word, depending on the 'num' argument. To facilitate this, the library comes with a set of 'rules' that specify the plural for some given nouns. If a user wants to extend the default rule set, they can use the `addRule` function. This function will modify the global rule set of the plural module and include the user's custom rules.

The existence of this function that can modify global state causes TestPilot to generate test suites with incompatible unit tests. Consider Listing 6.4, which shows the contents of test_6.js generated by cushman, one of the three completion models used in the TestPilot study [55]. Due to Mocha's alphabetical test order, it is always executed before test_9.js from the same test suite. Its contents are listed in Listing 6.5.

```
it('test case', function(done) {
  plural.addRule('test', 'test');
  done();
});
```

Listing 6.4: Adding a custom pluralization rule (test_6.js)

```
it('test case', function(done) {
  plural.unmonkeyPatch(); // This line is irrelevant
  assert.equal(plural('test'), 'tests');
  done();
});
```

Listing 6.5: Test expecting default behaviour (test_9.js)

After generation, TestPilot validates these tests independently by Mocha execution. Both tests pass and make it to the final test suite: Test 6 simply adds a rule to the module, and the test case passes because no exceptions are thrown; Test 9 succeeds because it correctly expects the plural of 'test' to be 'tests'. There is no explicit rule for the plural of 'test' in the module, so the plural function falls back to the general case: `return word + 's'`.

However, when these tests are executed consecutively, the rule added in test 6 will override this default fallback and test 9 will now fail, as its assertion now evaluates to `'test' == 'tests'`.

6.1.3 GPT-5.4 Results

The original TestPilot study evaluated older completion models, with GPT-3.5 Turbo performing best among the reported models. Because the agent in this thesis uses GPT-5.4, we perform an additional TestPilot replication with GPT-5.4 to create a stronger baseline. The GPT-5.4 replications will ultimately be used in a final comparison with the agent to answer RQ1.

Figure 6.1 compares the original GPT-Turbo TestPilot results with two GPT-5.4 TestPilot runs that follow the same generation approach under different output-token budgets: 500 tokens (GPT-5.4-500) and 1000 tokens (GPT-5.4-1000). Each row is one benchmark package; the gray dot is the GPT-Turbo baseline and the two green markers are the GPT-5.4 budgets, so the line between them shows the difference per LLM configuration. GPT-Turbo values are medians over 10 runs, while each GPT-5.4 value comes from

a single run. The left panel shows statement coverage and the right panel mutation score. The exact per-package values, including branch and line coverage, are listed in Table D.1 in Appendix D.

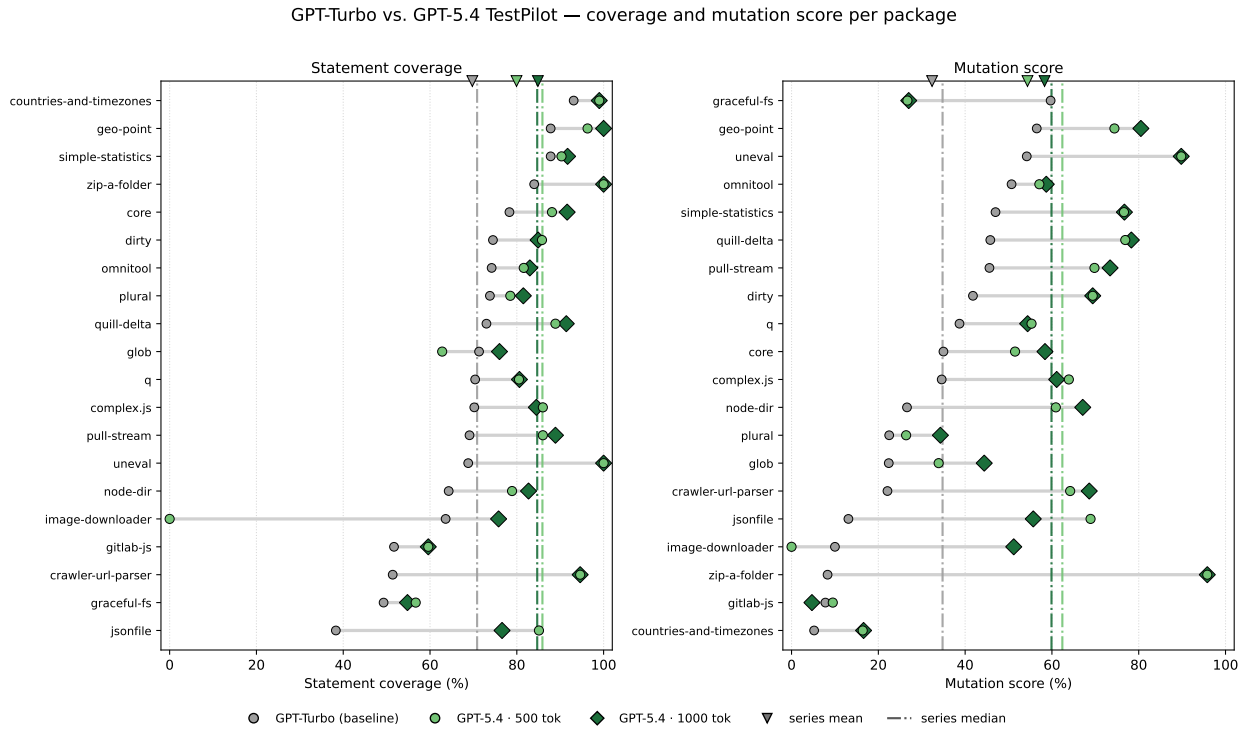

Figure 6.1: Per-package comparison of GPT-Turbo (best performing LLM originally studies by Schäfer et al. [55]) and the two GPT-5.4 TestPilot variants (500- and 1000-token output budgets) on statement coverage (left) and mutation score (right).

The GPT-5.4 variants substantially outperform the GPT-Turbo baseline on both metrics. The median mutation score increases from 34.8% for GPT-Turbo to 62.4% for GPT-5.4-500 and 59.9% for GPT-5.4-1000, a median per-project improvement of +21.0 and +27.0 percentage points respectively. Statement coverage improves similarly, with the median rising from 70.8% to 85.9% and 84.7%, for median per-project gains of +9.9 and +12.2 points. The dumbbell view shows these gains hold for a majority of the packages: the marker moves to the right for 18 of the 20 mutation scores, for 18 of 20 packages on coverage at the 500-token budget, and 20 of 20 at 1000 tokens.

There are some regressions, but they can mostly be attributed to natural variance and reporting only 1 run per GPT-5.4 configuration. The single GPT-5.4-500 run for `image-downloader` produced no passing tests (0% on both metrics) but recovers strongly at the 1000-token budget (+41.2 pp mutation, +12.1 pp coverage). `graceful-fs` is a filesystem-heavy package whose mutation score is unstable across reruns (discussed below), and at

the 500-token budget `glob` loses some statement coverage. `crawler-url-parser` originally scored 0% under both GPT-5.4 budgets, but we traced this to a TestPilot harness bug rather than a model failure. After applying a fix (discussed below) it becomes one of the larger gains.

In general, we see significant gains in both mutation score and statement coverage holding broadly across the benchmark. This suggests that to the extent our GPT-5.4-based agentic approach later outperforms the original TestPilot runs, a large part of that quality increase must be attributed to the improvement in underlying LLM capability alone.

The comparison between the two GPT-5.4 token budgets is inconclusive. GPT-5.4-500 has a slightly higher median mutation score and statement coverage score, while GPT-5.4-1000 has a slightly higher median branch coverage score and a larger median per-project improvement in both mutation score and statement coverage. Because each GPT-5.4 configuration was run only once, these small differences should not be interpreted as evidence that one token budget is generally preferable.

We do see that GPT-5.4-500 failed for image-downloader while GPT-5.4-1000 managed to achieve a 12.1pp increase over GPT-Turbo. We observe that for this case, 4 out of the 5 LLM conversations end because the output token limit was filled with reasoning tokens while generating no passing tests. The remaining prompt resulted in an incomplete test case that timed out. This is the strongest evidence we have for choosing one budget over the other. The median mutation and coverage scores are so close together that we are unable draw any other conclusions at N=1.

6.1.4 TestPilot and Stryker bug mitigations

Some bugs in the TestPilot test generation process only showed up during the GPT-5.4 run, but they were mitigated and/or fixed as follows.

Crawler-url-parser

Initially, neither of the GPT-5.4 runs produced any passing tests for crawler-url-parser. Our initial explanation was that crawler-url-parser is simply a difficult library for TestPilot: Schäfer et al. [55] report an average of 2 passing tests generated at best (GPT-turbo, cushman), and 1 at worst (starcoder) during the original run. However, an in-depth investigation into the GPT-5.4 experiments uncovered a minor harness bug.

TestPilot contains a post-processing cleanup step in which the generated test is passed through `fixMinorSyntaxIssues`, intended to repair small syntactic errors before the test is executed. One class of problems that is detected in this phase is an unbalanced number of

delimiters, like unclosed parentheses or missing '}' brackets. These problems are found and fixed by `closeBrackets`, a scanner that counts unmatched opening brackets and appends the necessary closing brackets before parsing the result. The scanner ignores line comments prefixed by '//', in order to avoid miscounting brackets that are just comments. However, the `closeBrackets` function can also apply this rule when string literals contain these characters. In URL literals such as http://..., the // are interpreted as delimiters for comments. The real closing brackets later on the same line are then skipped, and the bracket count turns out unbalanced. This results in otherwise valid GPT-5.4 tests being rejected due to a reported 'invalid syntax'.

When we reran the original completions through our fixed fork of the validator, we recovered 0 additional successful tests across the 30 runs. The original completion models did generate URL literals, but did not fail. The reason is that the original completion models generated simpler tests with no delimiters on multi-line URL literals containing // characters. A typical example is Listing 6.6, where `closeBrackets` would skip the remainder of lines 1 and 2 without missing any brackets or parentheses. On the other hand, the GPT-5.4-generated test in Listing 6.7 contains problematic lines with parentheses on the same lines as URL literals. This resulted in the validator detecting multiple unmatched opening parentheses (lines 1 and 4), and rejecting a perfectly fine test. We attribute the difference in generation styles mostly to a general increase in complexity of the model outputs enabled by the increased capabilities of GPT-5.4 over GPT-Turbo.

```
1 let linkurl = "http://www.baidu.com";
2 let pageurl = "http://www.baidu.com";
3 let type = crawler_url_parser.gettype(linkurl, pageurl);
4 assert.equal(type, "samelevel");
```

Listing 6.6: Excerpt from a test generated by GPT-5.4-1000

```
1   let r3 = crawler_url_parser.parse('../up/file.txt',
       'http://a.com/dir/sub/page.html');
2   assert.ok(r3);
3
4   let r4 = crawler_url_parser.parse('//cdn.example.org/lib.js',
       'https://site.test/app/');
5   assert.ok(r4);
```

Listing 6.7: Excerpt from a test generated by GPT-5.4-1000, incorrectly flagged for invalid syntax

Graceful-fs

GPT-Turbo originally achieved perfect mutation scores for graceful-fs, but in a suspicious manner: approximately 40 tests timed out during every regular test run. When Stryker re-executed the suite with mutants, all mutants covered by these tests are marked as timed out and killed. These cases are caused by flaky tests that kill large amounts of mutants. For example, test_113 generated by GPT-3.5 Turbo killed all 282 mutants in one run. This test is first run in the Stryker dry run, where it successfully creates a test directory and passes. Since the test does not contain any cleanup code, any subsequent executions will fail because the directory already exists. Stryker then finds that this previously passing test now fails for every possible mutant, and returns 100% mutation score.

```
it('test case', function(done) {
        let dir = __dirname + '/test_dir';
        graceful_fs.mkdirSync(dir);
    });
});
```

Listing 6.8: Test killing all mutants (test_113.js)

We re-ran the classic-model graceful-fs data through a filter that drops tests whose pass/fail depends on prior filesystem state, so a test can no longer mark every mutant as killed by failing identically everywhere. GPT-Turbo mutation score dropped to 59.7% (median), cushman to 59.1%, and starcoder achieved a mutation score of 10.7%.

6.1.5 Token Usage

Figure 6.2 summarises token usage for GPT-Turbo and both GPT-5.4 variants running through TestPilot's harness. The exact token counts are listed per library in Appendix Table D.2.

The GPT-Turbo values are estimated because TestPilot originally does not report on exact token counts. We use the persisted prompt to estimate the number of input tokens as prompt character count divided by four. Not all outputs have been recorded, so we cannot use the same estimation strategy for those. Instead, we assume an upper bound of the output token limit multiplied by the number of completions generated. The GPT-5.4 values are exact counts from the API, measured directly by our fork of TestPilot. The input and output columns of the appendix table show that GPT-5.4-500 and GPT-5.4-1000 spend roughly 3:1 output-to-input tokens, though GPT-5.4-1000 has higher absolute output volume. The estimation method means GPT-Turbo cannot be directly compared

to GPT-5.4, but it does reinforce our observation that TestPilot is output-heavy. These consistently high output-to-input ratios are logical considering TestPilot's approach: each function under test results in several independent completions, regardless of whether passing tests have been generated, or what coverage has already been achieved.

This prompting strategy also determines how much each library consumes in total: token usage scales roughly with project size, but not uniformly. Large libraries with many exported functions (omnitool, simple-statistics, graceful-fs) account for a disproportionate share of total tokens, since every public API member contributes at least one prompt-completion pair. Libraries with smaller API surfaces (zip-a-folder, jsonfile, uneval) consume comparatively fewer tokens.

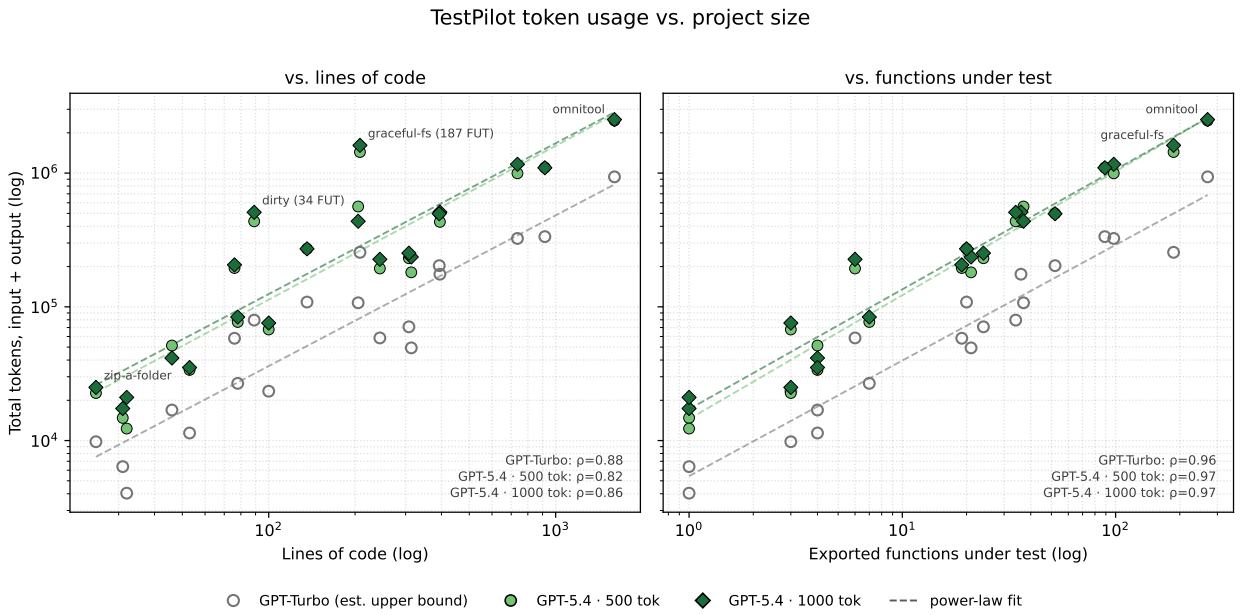

Figure 6.2: Total token usage (input + output) per library for GPT-Turbo and the two GPT-5.4 TestPilot budgets, against lines of code (left) and functions under test (right). Both axes are logarithmic, since library sizes and token counts each cover two orders of magnitude. Dashed lines are power-law fits, and ρ is the Spearman rank correlation between the size measure and token usage. GPT-Turbo Wvalues are estimated upper bounds (open markers). GPT-5.4 values are exact API counts.

How token usage scales with the amount of functions in the API is explicitly shown in Figure 6.2. It plots each library's total token usage against lines of code and the number of functions under test. For LoC, we see a moderate correlation of Spearman ρ between 0.82 and 0.88 across the three configurations. The biggest outliers are the libraries with relatively few lines per function: `graceful-fs` and `dirty`. When we plot token usage against the number of functions under test, we see a clearer trend ($\rho \geq 0.96$), supporting the idea that

TestPilot's token expenditure is a function of the number of API functions that are turned into prompts. These results are useful as a reference point when examining the agent's token usage in Section 6.2, where the balance between input and output tokens is inverted.

6.2 Agent Results

We present the results of the agent with access to all the features discussed in Chapter 5. The data in this section was produced by the two temperature 1 full-feedback runs, as discussed earlier in Section 4.5. We discuss failures (Section 6.2.1), the agent's mutation scores and how they relate to coverage (Section 6.2.2), and cost (Section 6.2.3).

6.2.1 Failed runs

Some libraries did not complete full measurements in both mutation-enabled runs. We distinguish generation failures from partial measurement failures. A run that produces zero passing tests contributes a 0.0% mutation score. A run that produces a passing suite is counted only when it yields a complete pair of measurements: the agent's own final `strykerMutationTest` score and the independent external benchmark that runs on the saved suite. When only one of the two is available, we drop that run from the library's cell, which then holds a single run's value rather than the mean of two. Such cells are marked with an asterisk in Appendix Table D.4 and all subsequent scoring tables and figures. This can happen when a test is flaky, causing it to pass one mutation test, but fail another. In practice, this only occurred for one run of graceful-fs, as reported below.

Both rules follow the same principle: we report what was correctly measured. An empty suite has a well-defined mutation score: it detects no mutants, so a generation failure contributes 0.0%. If we dropped this measurement, we would misrepresent information about a known unreliable result. By contrast, a run which passes the agent's own suite-wide mutation test but fails the external one, produces no real score: the agent-side measurement shows that the suite does detect mutants, so 0.0% would be false, but any other value also wouldn't be entirely correct. Therefore, we drop the unmeasured run and mark the cell, without hiding the failure itself. This means that the affected run is still counted as an agent validation failure in Table 6.3, so the reliability information does not disappear.

`uneval`. Both runs produced zero passing tests. The package has only one API item, so failing that results in no generated test suite. In one run, the agent first generated a passing test and passed a Stryker call, but introduced a wrong assertion in a later update. In the other run, the early-stop rule was activated after repeated `runTests` failures. In both cases repair then had no test file to run, Stryker was skipped, and the row has no score. These runs are included with scores of 0.0%.

`graceful-fs`. The first run completed the agent-side complete Stryker test, but the benchmark validation found one flaky failing test, causing it to skip the external Stryker benchmark. Because the full set of mutation scores was missing, we exclude this run. This was not a full generation failure, because 136 of the suite's 137 tests passed. Additionally, the agent-side score shows that the suite does detect mutants; only the second measurement of the pair is missing. The other run completed cleanly and supplies the starred `graceful-fs` scores.

`crawler-url-parser`. The first run produced no tests, even though every LLM request succeeded and the run terminated normally. All three public API items ended through the deliberate give-up mechanisms of the bounded-budget design (Section 5.2.5): one by the agent's own skip verdict, two by the early-stop rule after the same `runTests` failure repeated. In each case the model kept asserting wrong values for the library's unusual URL classification behaviour; the closest item had three of its four tests passing when the rule fired, but only fully passing test files are retained. This resulted in a correctly reported mutation score of 0.0%. The second run completed with a final score of 59.7%, resulting in a mean of 29.9% — best read as the expected outcome of a single agent invocation rather than a score either run achieved.

6.2.2 Mutation Score

The scatterplot in Figure 6.3 shows statement coverage and mutation score for the full mutation-enabled agent at temperature 1. It is the result of averaging two runs and scoring the failures as previously described.

The agent reaches a median statement coverage of 85% and a median mutation score of 76%, and most libraries sit in the upper-right region where high coverage is matched by a high mutation score. This is a good sign for the quality of the test suites generated by the agent: not only do they execute the code under test, the tests contain assertions strong enough to kill the mutants in that code.

A few libraries fall below this cluster. `uneval` is placed on the origin as the agent failed to produce tests both times (Section 6.2.1). `crawler-url-parser` is low because one of its replicate runs produced no tests, halving the averaged score. `countries-and-timezones` is an interesting outlier: it reaches reasonable coverage but a low mutation score. This is likely because its behaviour is largely a lookup table for timezones. The tests execute the lookup code, but don't actually reach all the entries, even though those are all mutated by Stryker. This is a clear example of the benefits of using mutation score alongside coverage as a test quality signal.

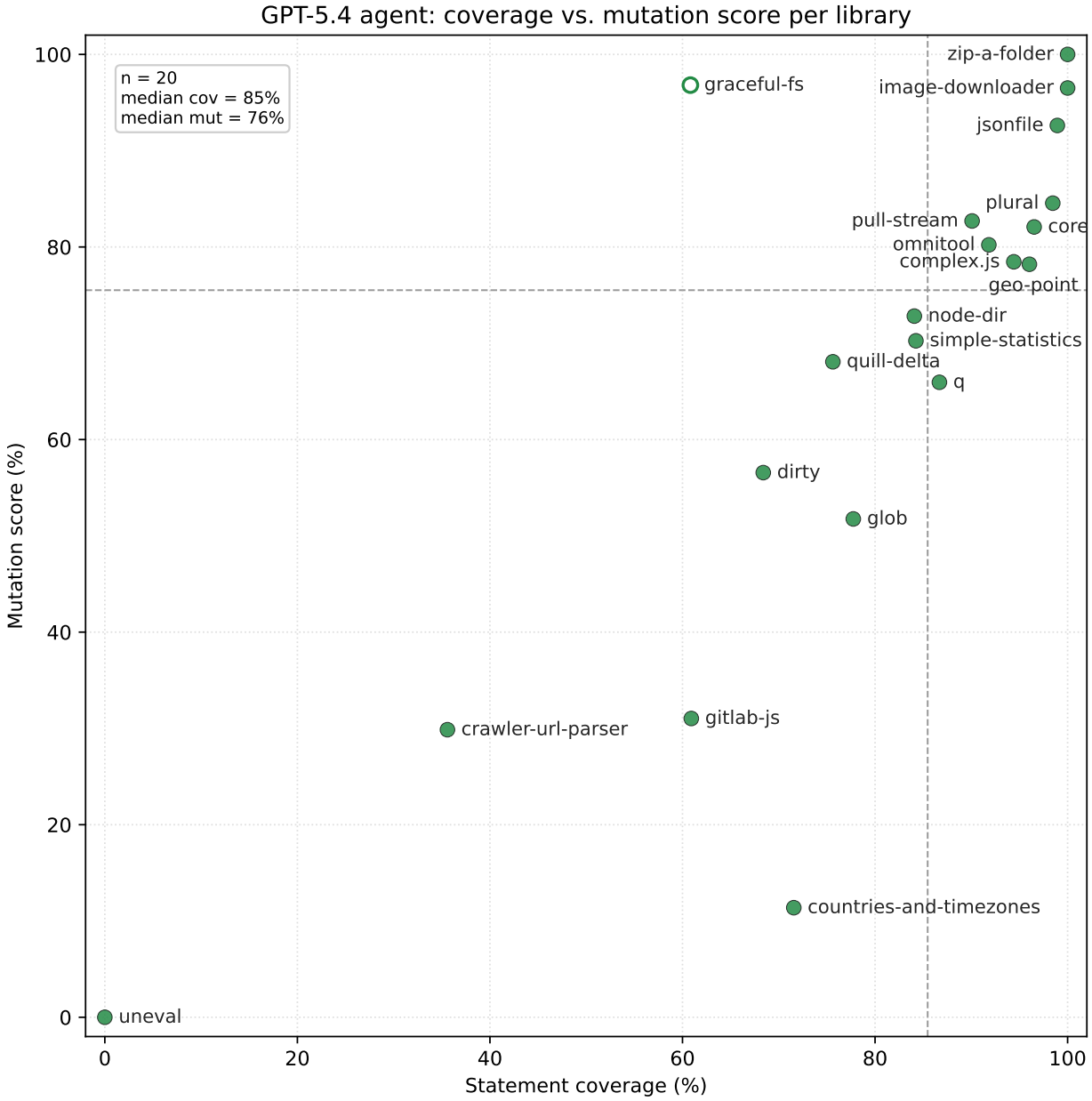

Figure 6.3: Per-library statement coverage versus mutation score for the GPT-5.4 agent. Each point is the average of 2 temperature 1 runs. No-test failures are scored as 0%, except for hollow markers: `graceful-fs` is a single-run value, the other mutation measurement was incomplete and omitted. Dashed lines represent the medians (85% coverage, 76% mutation score).

6.2.3 Cost

Table 6.1 contains the token usage, estimated API cost, request count, and generation time for the agent runs. Estimated cost is computed as described in Section 4.3, by pricing each run's tokens at the GPT-5.4 rate: non-cached input at $1.25, cached input at $0.125, and output at $10.00 per million tokens. The total row is computed by summing the means of all the libraries. It should therefore be read as the expected cost of one full 20-library run. This evaluates to 72.08M input tokens, 1.52M output tokens, 7,550 model requests, and an estimated cost of $32.34.

The token usage profile is opposite from TestPilot's, discussed in Section 6.1.5. TestPilot was output-heavy, with an input-to-output ratio of about 1:3. The agent instead spends input and output tokens at a rate of about 50:1. This is a result of the agentic architecture: we provide the agent with extensive context and tools to enhance its context gathering abilities, which help the LLM craft passing tests in fewer attempts. Additionally, the file writing tools allow tests to be rewritten by modifying only parts of the source, instead of the model having to regenerate the entire test. Still, the agent uses a significant amount of tokens, and prompt caching is what keeps this affordable. Since the prefix of each test generation item is largely the same, 89.9% of input tokens are cached and billed at the reduced cached rate, which is ten times cheaper than uncached input. Without caching, the same run would cost roughly $105 rather than the $32.34 reported above, more than tripling the price, so the cached prefix is what makes the agent's 50:1 input-to-output ratio affordable.

The cost distribution is also highly skewed. The median library costs only $0.68, but the mean is $1.62 because a few large libraries dominate the run: `omnitool`, `graceful-fs`, and `complex.js` together account for 59.3% of the estimated cost, and the top five libraries are responsible for 74.8%. These rows also have the largest request counts and runtimes: `omnitool` takes an average of 2,047 LLM requests, `graceful-fs` 1,404, and `complex.js` 946.

In total, the agent took about 11.9 hours of runtime, including Stryker and tool runtime. In larger projects, Stryker becomes the major bottleneck, especially during the residual runs. During the residual phase, the agent needs two more library-wide mutation tests, which is why `complex.js`, `graceful-fs`, and `omnitool` are expensive not only in tokens but also in time.

Table 6.1: GPT-5.4 agent: token usage, cost, and generation time, averaged across the two residual-mutation-feedback runs. Cells marked * were averaged from a single run.

Project	Input	Cached	Output	Cost (USD)	Requests	Time (min)
glob	1.26M	1.13M	40k	0.70	127	11.2
graceful-fs	15.51M	13.71M	353k	7.49	1,404	150.0
jsonfile	601k	548k	14k	0.28	72	4.2
q	5.65M	5.11M	130k	2.61	680	47.6
node-dir	1.14M	1.04M	36k	0.63	109	11.7
zip-a-folder	109k	99k	4k	0.07	16	1.7
quill-delta	2.47M	2.24M	62k	1.19	291	22.3
complex.js	9.38M	8.45M	181k	4.04	946	156.5
pull-stream	1.92M	1.75M	53k	0.95	226	21.8
countries-and-timezones	2.52M	2.31M	26k	0.82	124	13.1
simple-statistics	5.37M	4.84M	114k	2.40	638	53.0
plural	654k	589k	13k	0.29	78	4.7
dirty	628k	568k	19k	0.34	64	5.5
geo-point	1.21M	1.09M	31k	0.60	140	8.8
uneval	130k	118k	4k	0.07	14	1.0
image-downloader	370k	338k	10k	0.18	40	3.0
crawler-url-parser	425k	388k	10k	0.20	44	3.1
gitlab-js	2.68M	2.42M	54k	1.17	318	18.7
core	1.42M	1.30M	34k	0.66	173	10.3
omnitool	18.63M	16.80M	327k	7.66	2,047	165.2
Total	72.08M	64.83M	1.52M	32.34	7,550	713.5

6.3 Agent vs. TestPilot

This section provides a side-by-side comparison of the results achieved by TestPilot with GPT-5.4 and the results of our agent on the same model, across mutation score, coverage, correctness, test non-triviality, and cost.

Unless noted otherwise, the agent figures in this section come from the two full-feedback temperature 1 runs also used in Section 6.2.

6.3.1 Mutation score, coverage, and passing tests

Table 6.2 compares the averages of metrics that are relevant for both systems. The per-library data is shown in Figure 6.4 and listed in full in Appendix Table D.3.

The agent achieves a higher average mutation score than both GPT-5.4 TestPilot replications: a mean mutation score of 66.5% and a median of 75.5% compared with 54.4%/62.4% for GPT-5.4 TestPilot-500 and 58.3%/59.9% for GPT-5.4 TestPilot-1000. The agent scores higher than GPT-5.4-500 on 14 of 20 libraries and outscores GPT-5.4-1000 on 13 of 20.

Table 6.2: Comparison between GPT-5.4 TestPilot and the final GPT-5.4 agent. *Mut.* is mutation score (mean/median over libraries); *Cov.* is coverage (statement/branch); *Tests* and *LoC* are the passing test count and the lines of passing test code; *In*/*Cached*/*Out* are token totals. *Min/lib* is mean minutes per library.

System	Mut.	Cov.	Tests	LoC	In	Cached	Out	Cost	Min/lib
TestPilot-500	54.4%/62.4%	79.9%/68.7%	3,057	84.7k	2.38M	0	6.91M	$72.09	24.7
TestPilot-1000	58.3%/59.9%	84.9%/74.3%	3,246	94.2k	2.61M	0	7.22M	$75.47	29.2
Agent	66.5%/75.5%	78.6%/71.3%	826	40.3k	72.08M	64.83M	1.52M	$32.34	35.7

The coverage results are closer, and here TestPilot is in fact slightly ahead. The agent reaches 78.6% mean statement coverage and 71.3% mean branch coverage after no-test runs are counted as 0% coverage, compared with 79.9%/68.7% for TestPilot-500 and 84.9%/74.3% for TestPilot-1000. The agent's statement coverage is therefore marginally below both TestPilot budgets, even though its mutation score is substantially higher. This means the agent kills more mutants while executing a comparable or smaller amount of code, which indicates that it concentrates stronger assertions on the code it does cover rather than simply reaching more of it. This pattern is directly visible in Figure 6.4: in the mutation panel the three median lines separate clearly, while in the coverage panel the three series' summary marks nearly coincide.

TestPilot is still stronger on several libraries, visible as the bottom rows of the mutation panel in Figure 6.4. Both GPT-5.4 variants remain ahead on `quill-delta`, `countries-and-timezones`, `simple-statistics`, `dirty`, `uneval`, and `crawler-url-parser`. The 1000-token budget also remains ahead on `geo-point`. The `uneval` and `crawler-url-parser` losses are partly caused by the agent's no-test failures, while the other losses show that the agentic strategy is not uniformly better. For small or straightforward APIs, TestPilot's multiple completion strategy can be sufficient.

The solutions also differ in the number of tests they produce. The two GPT-5.4 TestPilot experiments produce 3,057 and 3,246 passing tests respectively, while the final agent produces an average of 826 passing tests across one full 20-library run. The agent therefore achieves higher mutation scores with far fewer tests.

When we look at the total size of the test suites produced by both systems, we see that the agent does produce longer tests, but not enough to negate the difference. In total, the agent produces an average of 40k lines, versus 85k for GPT-5.4 TestPilot-500 and 94k for GPT-5.4 TestPilot-1000. This is less than half the volume, with the individual agent tests being about 49 lines per passing test, versus 28–29 for TestPilot. The agent is able to concentrate more powerful assertions in less code, reaching a higher mutation score while producing under half the test code.

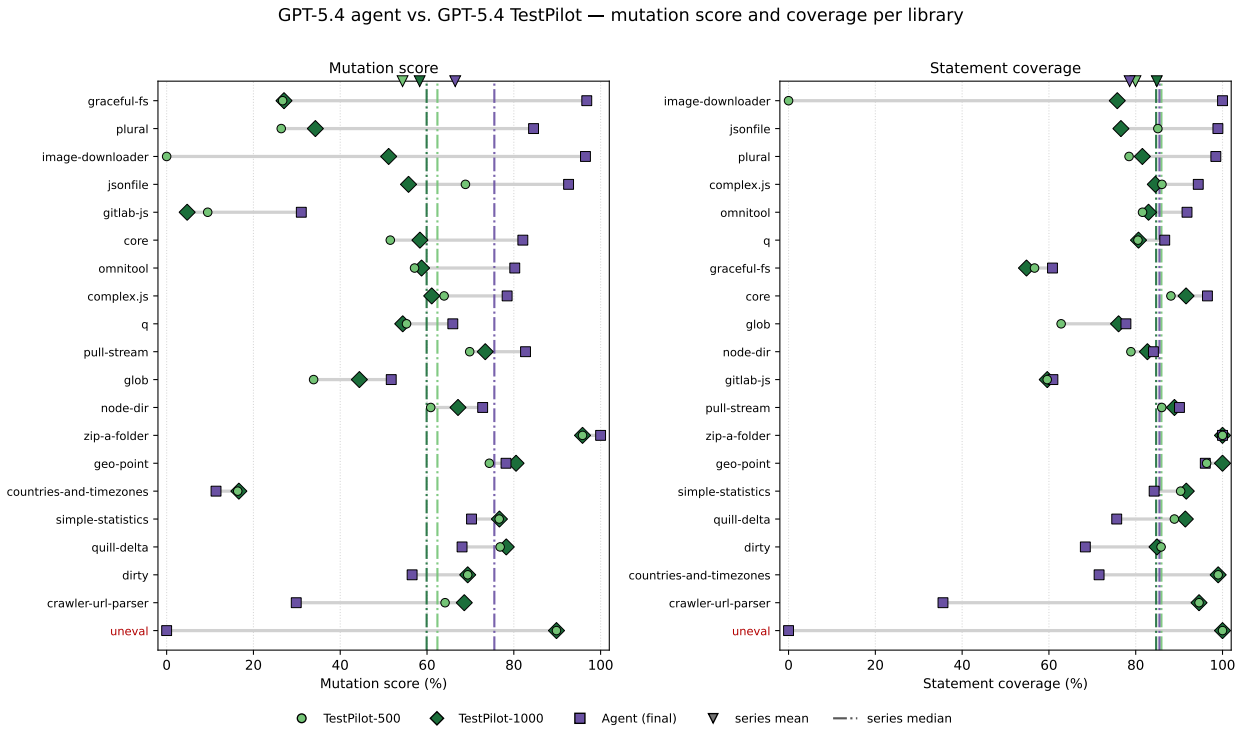

Figure 6.4: Per-library mutation score (left) and statement coverage (right) for the two GPT-5.4 TestPilot budgets and the final agent. Rows are sorted by the agent's difference with TestPilot-1000. Vertical lines mark each the median and the carets on the top edge mark the means. `uneval` (red label) is an agent no-test failure in both runs, scored as 0.0%.

6.3.2 Correctness

To combat some of TestPilot's replication issues discussed earlier in Section 6.1.2, the agent validates the tests at two levels. First at the API item level, where the tests for an item must pass `runTests`, a simple tool that just calls the Mocha test runner, before being allowed to request `strykerMutationTest`. On the suite level, the harness initiates repair phases when there are failures in the full suite.

Table 6.3: Number of runs affected by validation failures. "Validation failures" are attempts that produced tests but failed later validation. "T0" and "T1" refer to the agent experiments at temperature 0.0 and 1.0, respectively.

System	No passing tests	Validation failures	Total affected
TestPilot-500	1/20 (5.0%)	4/20 (20.0%)	5/20 (25.0%)
TestPilot-1000	0/20 (0.0%)	6/20 (30.0%)	6/20 (30.0%)
Agent full+residual T1	3/40 (7.5%)	1/40 (2.5%)	4/40 (10.0%)
Agent full+residual T0	0/20 (0.0%)	0/20 (0.0%)	0/20 (0.0%)

Table 6.3 contains the correctness data for both systems. We see that the rate at which they fail to produce any passing tests at all is comparable. TestPilot produced no passing tests for 1 of 20 libraries in the GPT-5.4-500 run (5.0%) and for 0 of 20 libraries in the GPT-5.4-1000 run (0.0%). The temperature 1 mutation-enabled agent has no retained tests in 3 of 40 library-runs (7.5%). The temperature 0 agent did not produce any outright failures, but was only executed for one set of the 20 libraries.

There is a clear difference in failures exposed by suite-wide validation. Beyond the outright failures, TestPilot had 4 further libraries for GPT-5.4-500 and 6 for GPT-5.4-1000 whose tests failed the mutation dry run, giving a total of 5 affected libraries out of 20 for GPT-5.4-500 (25.0%) and 6 out of 20 for GPT-5.4-1000 (30.0%). By contrast, the mutation-enabled agent exposed only 1 such suite-validation failure across 40 library-runs at temperature 1 (2.5%) and none at temperature 0, so its total affected rate is 4 out of 40 (10.0%) at temperature 1 and 4 out of 60 (6.7%) across all mutation-enabled runs.

This indicates that the agent's validation and repair loop makes generated suites more robust. However, it does not solve suite contamination completely. One agent run without mutation feedback (not included in Table 6.3) still contained a contamination case for the problematic `plural` library. The agent therefore reduces the rate of affected runs, but is not completely immune to this failure mode. Besides, other unobserved failures can still remain.

6.3.3 Non-Trivial Tests

The TestPilot harness measures whether generated tests contain *non-trivial assertions*: assertions which can be traced backwards to reach the import statement of the package under test. For GPT-5.4, TestPilot-500 produced non-trivial tests at a rate of 92.3% of all generated tests and TestPilot-1000 at 92.7% (Table 6.4).

Running the original TestPilot classifier unchanged on the agent suites marks all agent tests as trivial. This is not because the tests lack assertions, but because the agent uses different import conventions than TestPilot. The original query expects Node's unprefixed `assert` module and a package-name import such as `require("q")`. The agent tests usually import Node's `node:assert` variants and import the checked-out package by relative path, as in Listing 6.9. We therefore add an agent-compatible criterion that also accepts `node:assert`, `node:assert/strict`, and relative imports that reach the archived package under test.

Table 6.4: Non-triviality of generated tests. *Pass* is the share of generated tests that pass. NT^* is the percentage of all generated tests that contain a non-trivial assertion, and NT^* (pass.) the same percentage but for passing tests only. TestPilot and GPT-Turbo use the original criterion, while agent rows use the agent-compatible metric.

Generator (condition)	Runs	Tests	Pass	NT*	NT* (pass.)
GPT-Turbo (original)	10	4,984	42.3%	71.2%	67.8%
TestPilot-500	1	8,872	34.5%	92.3%	97.2%
TestPilot-1000	1	9,357	34.7%	92.7%	97.4%
Agent, full+residual (T=1)	2	827	99.9%	99.1%	99.1%
Agent, hidden (T=1)	2	821	99.9%	99.6%	99.6%
Agent, full+residual (T=0)	1	848	100.0%	98.1%	98.1%

*Re-scoring the TestPilot/GPT-Turbo suites under the agent-compatible criterion does not change their non-triviality score ($\leq$0.02 pp; zero affected files for GPT-5.4), so the difference does not give either system an advantage.

```
const assert = require('node:assert/strict');
const Complex = require('../../complex.js');

it('matches Math.acos for real inputs', function () {
  const value = new Complex(0.5, 0).acos();
  assert.strictEqual(value.im, 0);
});
```

Listing 6.9: Agent test pattern detected by the updated non-triviality query

Under the agent-compatible definition, the agent suites are almost entirely non-trivial: Table 6.4 shows non-triviality rates (NT^*) of 99.1% for the mutation-enabled runs, 99.6% for the hidden runs, and 98.1% for the temperature 0 run. Meanwhile, the 2 GPT-5.4 TestPilot budgets score 92.3% and 92.7%.

For the non-triviality of passing tests, the agent produces effectively the same number because almost every generated test also passes. For TestPilot roughly a third of TestPilot's generated tests are passing. GPT-5.4 TestPilot therefore already produces highly non-trivial tests, but the agent suites are both almost entirely non-trivial and almost entirely passing.

We do not report non-trivial coverage scores for the agent, because the agent reports do not retain the per-test coverage needed to recompute that metric without rerunning the suites.

6.3.4 Token costs and runtime

When we compare the token costs discussed in sections 6.1.5 and 6.2.3, we see that both approaches come with dramatically different token usage profiles. TestPilot produces output tokens at thrice the rate it consumes input tokens, while our agent uses approximately 50 input tokens for every output token. This follows from the differences in design: TestPilot samples multiple LLM outputs, while the agent injects extra context in its input using tool calls. Additionally, the TestPilot tokens are not eligible to be cached due to the small size of the per-item prompts, while the agent can take full advantage of the reduced cached token prices (Section 4.3): its starting prompt is larger because it must contain detailed tool instructions and parameters, and every per-item conversation has many turns that incrementally grow that cached prefix. Under this pricing, this makes the final agent cheaper in estimated API cost ($32.34) than TestPilot-500 ($72.09) and TestPilot-1000 ($75.47), despite the agent's much larger raw input-token count.

TestPilot does beat the agent in runtime. A full residual agent run takes about 11.9 hours. TestPilot achieves a total runtime of about 8.2 and 9.7 hours, which makes the agent slower, but cheaper.

6.3.5 Overall performance

These comparisons show a consistently positive picture during our evaluation: on the 20 GPT-5.4 libraries, the agent produces suites that are almost entirely valid, kill more mutants (66.5% against 54.4–58.3%), cost under half as much to generate ($32.34 against $72–$75), and are less than half the size (on average 826 passing tests and 40k lines of test code, against roughly 3,000–3,200 tests and 85–94k lines). TestPilot's suites only achieve a slightly higher statement coverage and a shorter runtime.

However, these results hold only for GPT-5.4 on a sample of 20 libraries with two runs per condition. Further limitations and the extent to which we can generalize are discussed in the next chapter.

6.4 Temperature Sensitivity

To check whether the agent's results depend on decoding temperature, we ran one additional full benchmark suite at temperature 0. This run uses the same agent commit as the other runs with full mutation feedback. We compare it with the two temperature 1 residual runs used in the previous tables. Temperature 0 has only one run, but we can expect the variance to be very low at this temperature. That is why we can compare it against 2 temperature 1 runs.

6.4.1 Effect of temperature on mutation score

The results are available in Figure 6.5. On average, they do not show a clear temperature effect on mutation score; most libraries barely move between the two settings and the extremes cancel out. Temperature 0 ends up with a slightly higher mean score both for partial feedback (58.5% vs. 58.1%) and after residual feedback (68.2% vs. 66.5%). However, these differences are strongly affected by temperature 1 no-test failures: `uneval` produces no tests in either run (counted as 0.0%), and one of `crawler-url-parser`'s two runs also produced no tests, lowering its temperature 1 mean to 29.9%. The paired differences are much smaller, with the median delta −0.5pp before residual feedback and −0.1pp after.

We can discern a pattern separating the sign of the delta: The libraries that that do better for temperature 1 (`jsonfile` and below) almost all use I/O or are stateful or asynchronous, while the temperature 0 improvements are mostly pure computation libraries like `simple-statistics`, `geo-point`, `q`, and `core`. However, it is difficult to say if this is a real temperature effect.

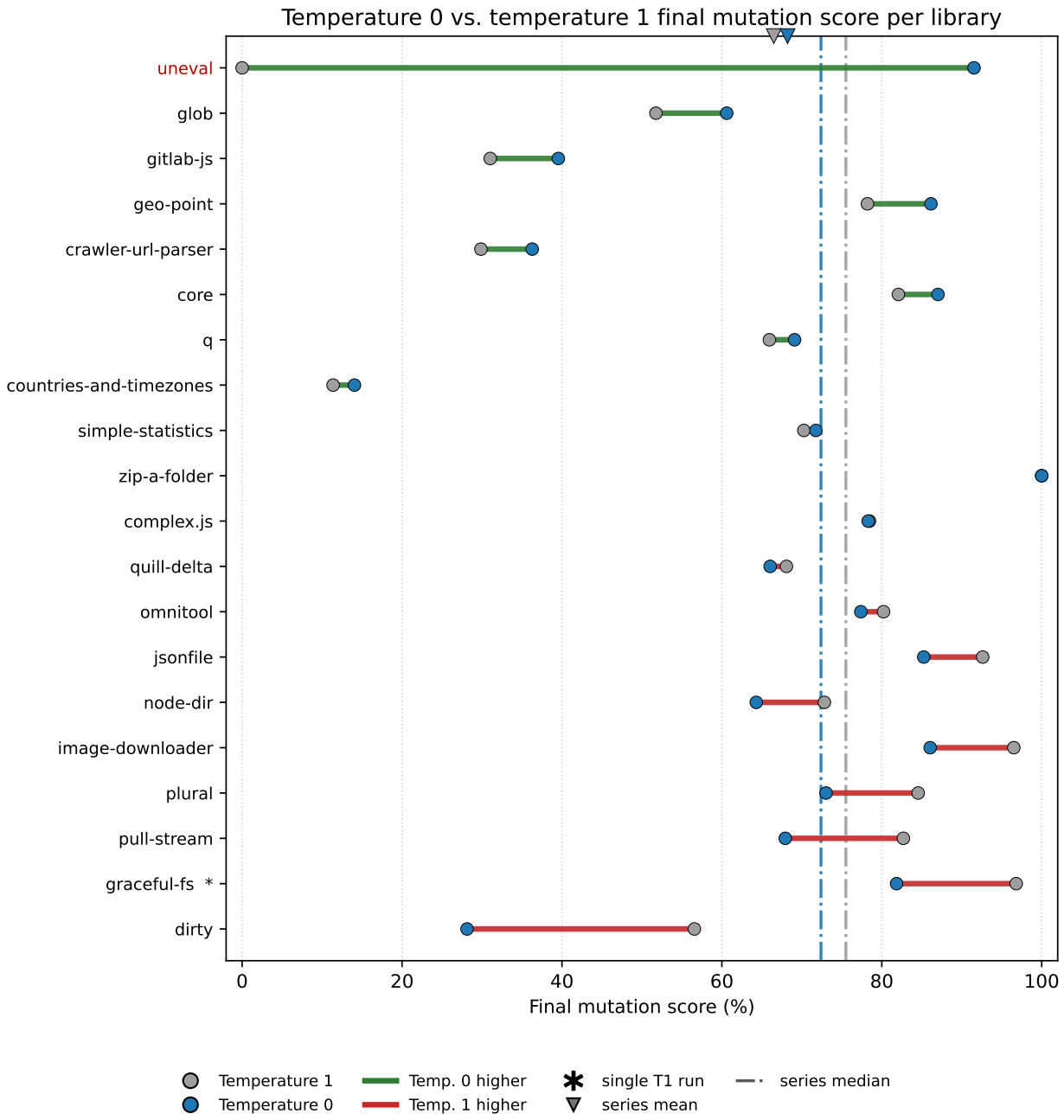

Figure 6.5: Final (post-residual) mutation score at temperature 0 versus temperature 1, sorted by the temperature difference. The line colour shows direction (green: temperature 0 higher; red: temperature 1 higher). `uneval` (red label) produced no passing tests in either temperature 1 run.

The I/O-heavy libraries at the temperature 1 end are also the ones with the highest run-to-run variance (Section 2.3.3), so their large deltas could just be noise instead of temperature effects, especially since temperature 0 is a single run compared against a two-run temperature 1 mean (and `graceful-fs` is a single run at both settings).

The split is also not entirely clean: The largest temperature 0 improvement, `uneval`, is caused by 2 failing temperature 1 runs. Additionally, the I/O-heavy `glob` and `gitlab-js` fall on the temperature 0 side. We therefore cannot interpret this pattern as a systemic temperature effect.

6.4.2 Effect on completion

The clearest difference is the number of libraries for which a full test suite is completed. The temperature 0 agent completed and scored all 20 libraries, with no partial or failed libraries. By contrast, the two temperature 1 runs include no-test outcomes for two libraries: `uneval` fails in both runs, and `crawler-url-parser` fails in one run. At temperature 0, `uneval` reaches 91.5% and `crawler-url-parser` improves from 23.9% pre-residual to 36.3% final.

6.5 Ablation study for mutation feedback

Mutation analysis reaches the agent at two distinct points, and we evaluate each separately. During the per-item generation loop, every successful `strykerMutationTest` call returns mutation feedback that the model uses to only modify the tests for the API item it is working on. At the end of the run, the residual phase selects aggregate survivor clusters across the whole library, and adds them as artificial API items for which the agent generates additional targeted tests (Section 5.3.6).

6.5.1 Fine-grained feedback

We measure the results achieved by the agent in two runs with per-item mutation feedback disabled. In this hidden mode, the agent contains the exact same instructions as normal but every successful `strykerMutationTest` call returns "Mutation test completed. Successful mutation feedback is hidden for this ablation condition." If `strykerMutationTest` fails, the model still gets access to the error messages to repair any invalid tests. This gives the hidden-mode agent the exact same context as the mutation-enabled one, without access to mutation scores and mutant information.

Note that the residual phase necessarily includes mutant information, as described in Section 5.3.6. This means that this entire phase is unavailable in hidden mode. However, we can still provide a clean ablation study by simply comparing the final mutation scores achieved on hidden mode to the pre-residual scores achieved by the mutation-enabled agent. This allows us to examine only the effects of mutation feedback inside the per-item test generation loops. Figure 6.6 presents the resulting comparison.

The ablation does not show a clear positive effect from adding per-item mutation feedback to the agent. The mean Δ_{feedback} is -3.2pp and the median is -1.7pp. If mutation feedback were effective, we would expect to see positive values. Full mutation feedback outperforms hidden feedback on only 7 libraries, hidden feedback scores higher than full feedback on 13 libraries, and no library is tied. The result therefore does not support a reliable positive effect from exposing per-item mutation feedback to the model.

In fact, there are some large differences in favour of leaving out mutation feedback. For example, `uneval` and `crawler-url-parser`, where the full-feedback condition includes no-test failures under the scoring rule described above. However these are not direct evidence that hidden feedback is generally better. They are more plausibly explained by run-level variation combined with the fact that iteration on mutation feedback provides an extra failure possibility. In the current agent, a passing test might end up failing after the agent

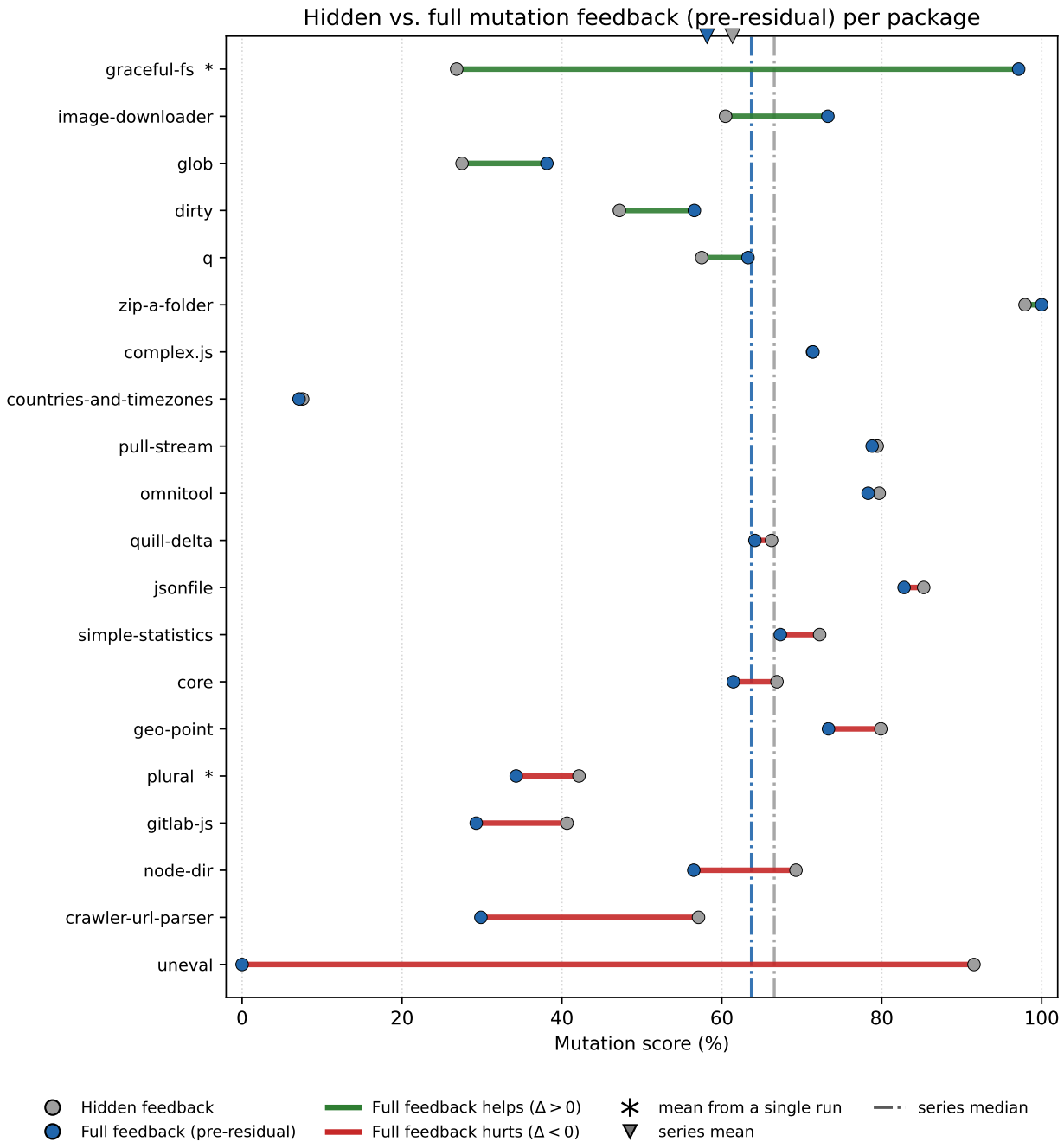

Figure 6.6: Hidden vs. full mutation-feedback ablation (pre-residual) per package. Rows are sorted by Δ_{feedback}. Rows marked $*$ used a single run for a mean.

attempts to incorporate mutation feedback. One hidden run for `node-dir` created passing tests for `promiseFiles`, `files`, `readFiles`, and `readFilesStream`. In the mutation-enabled run, several of these items generated successful tests that were broken by modifications after `strykerMutationTest` feedback. Across the two mutation-enabled runs, we found 228 passing test cases that could have been preserved by rolling back after a failed mutation-

feedback iteration. These discarded tests amount to 13.8% of all tests written across the two mutation-enabled runs.

6.5.2 The residual phase

Figure 6.7 shows the mutation scores before and after the residual phase, at both decoding temperatures (the exact per-library scores and differences are listed in Appendix Table D.4). The *pre-residual* score is the agent-side Stryker score after the public-API generation loop and repair phase; *agent final* is the score after the residual phase has completed, and Δ is their difference. Temperature 1 is the mean of two runs and temperature 0 a single run. As elsewhere, agent failures with zero passing tests are included as 0.0% scores, while other incomplete measurements are omitted, and the affected values are marked $*$.

In contrast to the per-item signal, the residual phase strictly increases the mutation score on most libraries. Across the 20 libraries at temperature 1, 15 improve, 4 are unchanged, and 1 decreases. The mean difference is +8.4 percentage points and the median difference is +3.9 percentage points. The median mutation score rises from 63.7% pre-residual to 75.5% after residual feedback. This gain is not specific to temperature 1: the temperature 0 panel of Figure 6.7 shows a comparable effect (+9.7pp mean, +6.3pp median, with 16 of 20 libraries improving), so the residual phase helps regardless of decoding temperature.

We see the biggest improvements on libraries where the API loop leaves concentrated survivor clusters untested. For example, `plural`, which motivated the addition of the residual phase (see Section 5.2.6), improves by +50.3pp. Similarly, `image-downloader` increases by +23.3pp, `core` by +20.6pp, `node-dir` by +16.3pp, and `glob` by +13.6pp. Smaller gains on libraries like `quill-delta` (+3.9pp), `gitlab-js` (+1.8pp), and `simple-statistics` (+2.9pp) show that residual feedback can still be useful after a strong pass with just the API-guided generation, but with diminishing returns.

The four unchanged libraries have different causes. `zip-a-folder` already reaches 100.0% before the residual phase, leaving no room for improvement. `dirty` still had survivor clusters after the public-API pass, but the residual planner selected none of them: in both runs, all available clusters overlapped source ranges that had already been assigned to public-API items. `uneval` is unchanged because both runs are no-test failures scored as 0.0%. `crawler-url-parser` averages one no-test failure with one completed run. In the completed run, the residual phase selected two clusters, but neither changed the final score: one targeted an unused object, and the other demo code guarded by `if (!module.parent)`, which generated Mocha tests cannot cover when they import the library.

`graceful-fs` is the only library with a negative residual delta, decreasing by 0.3pp in the

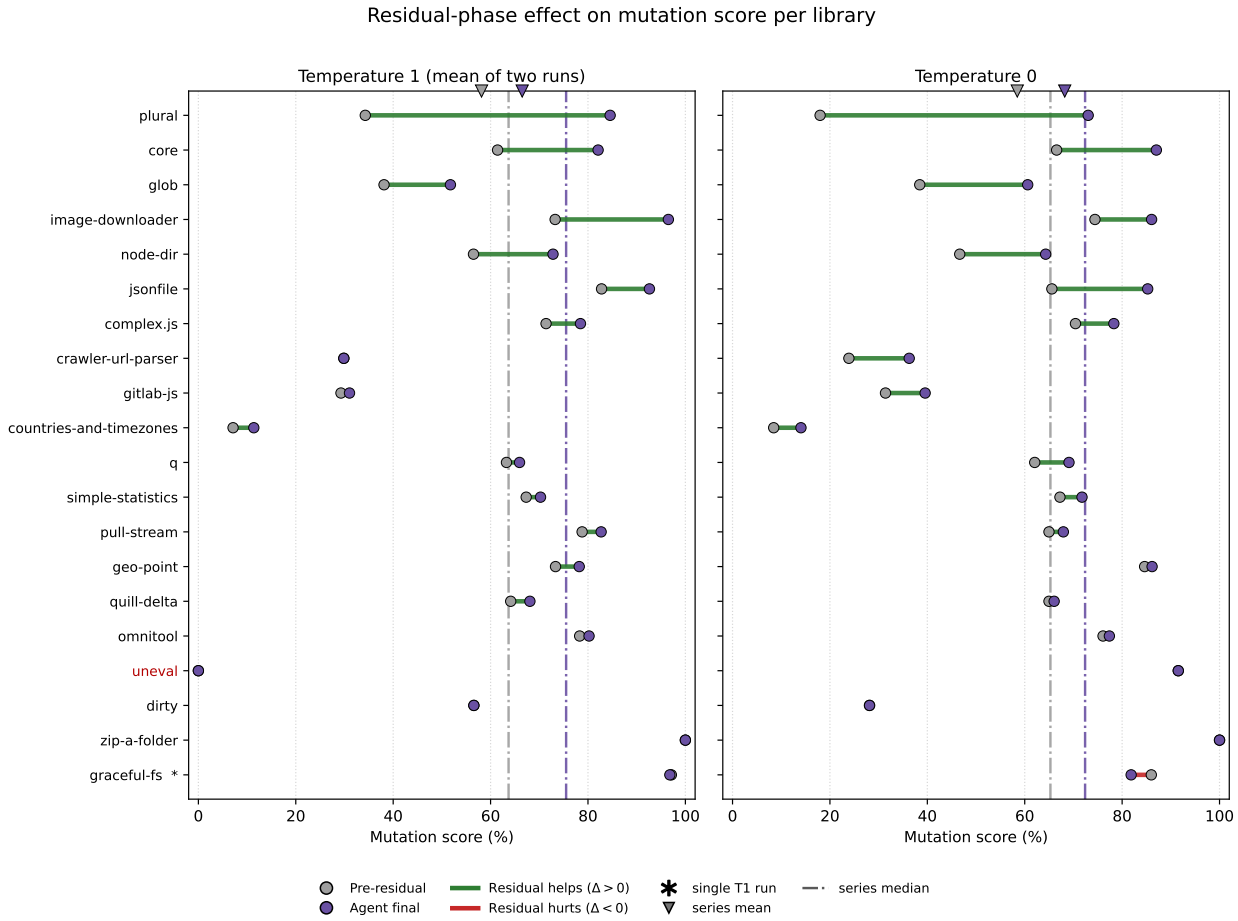

Figure 6.7: Mutation score before and after the residual phase, per library and decoding temperature. The temperature 1 panel shows the mean of the two temperature 1 runs. Rows are sorted by the mean residual gain across both temperatures. Green segments represent libraries the residual phase improves and red segments the single case it lowers (`graceful-fs` at temperature 0). Vertical lines represent medians carets on the top represent means. `uneval` (red label) produced no tests in either temperature 1 run and is scored 0.0% there. The asterisk marks `graceful-fs`'s single-run temperature 1 value.

single completed score run. We attribute the difference to natural variance in mutation scores for larger test suites. For `graceful-fs`, the new residual tests execute "polyfill behaviour" (i.e. code that normalizes platform features so that graceful-fs presents a consistent API across Node versions), like module loading and the require cache. These tests can interact with Stryker's Mocha workers by changing module initialisation order, and therefore timeout behaviour. Since Stryker counts timeouts as detected mutants, a mutant that was previously killed or timed out can later complete and survive when the suite composition changes.

The residual phase produces a consistent gain, although at a small token cost. When we measure LLM requests within the residual phase, residual generation accounts for 11.9%

of the agent's tokens (about 8.8M of the 73.6M per run) and 11.8% of the estimated API cost (roughly $3.8 of the $32.34 per run). Against the +8.4pp mean (+3.9pp median) improvement, it is a token-efficient source of gains. The real cost is runtime: the two extra library-wide Stryker runs it triggers are the main runtime bottleneck (Section 6.2.3), not its tokens.

Together, these two effects show how the effect of mutation feedback can differ based on how it is inserted. Exposing per-item mutation feedback during generation does not reliably improve scores, and can even corrupt previously passing tests. The project-wide residual phase however, is responsible for a large part of the agent's higher mutation scores, despite using relatively few tokens.

CHAPTER 7: DISCUSSION

This chapter will interpret the experimental results with regard to the research questions posed in Chapter 4. Additionally, we will provide an analysis of the limitations and possible threats to validity of our research. Finally, we will provide recommendations for future work, discuss the practical applicability of our agent, and conclude the thesis.

7.1 Answering the Research Questions

This section answers the two research questions from Chapter 4:

> **RQ1: How effective is an MCP-based agentic test generation system for generating JavaScript unit tests?**
>
> **RQ2: How does incorporating mutation testing feedback through StrykerJS affect the quality of LLM-generated tests?**

The validation questions are used as supporting questions that describe the metrics along which we evaluate the RQs:

> **VQ1.1: How does the correctness, coverage and mutation score of tests generated by the agentic approach compare to those generated by TestPilot?**
>
> **VQ1.2: How does the token and runtime cost of the agentic approach compare to TestPilot's light context test generation baseline?**
>
> **VQ2.1: How does mutation feedback targeted to the API item level affect project-wide mutation score?**
>
> **VQ2.2: How does the residual mutation feedback phase affect mutation score relative to its additional token overhead?**

7.1.1 RQ1: Effectiveness of the Agentic Test Generation System

RQ1 asks how effective our agentic test generation system is for generating JavaScript unit tests. We can use VQ1.1, which investigates test correctness, coverage, and mutation score, to formulate an answer. In the comparison with the GPT-5.4 TestPilot baseline, we see the following:

- the agent generates fewer tests, of which more are correct
- the agent achieves slightly lower statement coverage on average
- the agent achieves a higher mutation score on average

The comparison in Table 6.2 shows that the agent's clear advantage is in mutation score, while its statement coverage is in fact slightly below both TestPilot budgets. This is useful information because coverage only measures whether code is executed, while mutation score also depends on whether the tests assert behaviour strongly enough to detect changes. The agent therefore gains its advantage not by executing more code, but by generating more effective assertions for the code it does execute. We also see that the agent's tests suites are smaller: they are under half the LoC of TestPilot's.

For correctness, as discussed in Section 6.1.2, TestPilot validates generated tests independently, which allows some tests to pass in isolation but fail when executed as a full suite. The agent instead validates tests both at the API item level and at the test suite level, with repair phases between generation and final evaluation. This does not eliminate the correctness problems entirely, but it reduces the number of affected runs and makes failures more visible in the generated reports.

VQ1.2 qualifies the answer by considering cost. The agent is cheaper in estimated API cost under the measured GPT-5.4 pricing setup, largely because prompt caching reduces the price of its large repeated input contexts. The agent spends many conversation turns interacting with tools and StrykerJS runs. These are all opportunities for taking advantage of token caching, but the tool calls do increase the runtime beyond TestPilot's.

Overall, RQ1 can be answered positively: our MCP-based agentic system is an effective JavaScript unit test generator in this benchmark. Its main strengths are free context exploration, iterative execution feedback, suite-level repair, and mutation-guided residual targeting. Its main weakness is runtime, especially on large libraries with many API items or expensive mutation runs.

7.1.2 RQ2: Effect of Mutation Testing Feedback

RQ2 asks how incorporating mutation testing feedback through StrykerJS affects the quality of LLM-generated tests. We find that the answer depends on how mutation analysis is incorporated. The per-item mutation feedback ablation does not show a reliable positive effect from exposing this small-scoped feedback directly to the model. The residual feedback phase however, which also depends on mutation analysis, does improve mutation score consistently.

Our explanation for the nonexistent effect of per-item mutation feedback is that it introduces an extra failure opportunity. As discussed in Section 6.5.1, 13.8% of all tests written across both mutation-enabled runs only fail after the attempted integration of mutation feedback.

It is possible to conclude what the benefits of per-item mutation analysis would be without this drawback. Currently, the tests for an API item are committed or discarded based on their pass/fail once all exploration and mutation analysis has been completed. The introduction of a rollback mechanism that would restore tests to their last successful state inside the per-item generation process would support that analysis. We reserve this for future work.

When we analyse the effect of mutation analysis according to VQ2.2, which concerns the residual phase, we see a different result. Section 6.5.2 reports consistent improvements in mutation score after the residual phase (+8.4pp on average). Unlike per-item feedback, it avoids the test-regression problem: the residual phase only adds tests, so failing additions are simply discarded and cannot lower the score (the single small negative, `graceful-fs`, is Stryker measurement noise rather than a regression). It is also cheap relative to its benefit, costing only about 12% of the run's tokens. This directly answers VQ2.2: the residual phase improves mutation score at a small token overhead.

RQ2 should therefore be answered in two parts: We find that mutation feedback scoped to the unit level does not directly improve test quality, but that mutation analysis can improve a test suite when it is used to point an agent at suite-level weak points.

7.2 Threats to Validity

This section discusses threats to the validity of the experimental results. Following the classification of Wohlin et al. [69], we distinguish conclusion validity, internal validity, construct validity, and external validity.

Conclusion validity refers to whether the amount and variability of the evidence are sufficient for the claims we make. Internal validity concerns whether our observations can actually be attributed to the experimental conditions. Construct validity regards whether the selected metrics measure the intended concepts. Finally, external validity concerns whether the results can generalize beyond the benchmarks, model, and tooling used in this thesis.

Threats to Conclusion Validity

The main conclusion validity threat is nondeterminism in the generation and evaluation pipeline. Even when the same agent configuration is used, the generated tests can differ

between runs. In addition, StrykerJS mutation scores are not always perfectly stable between runs. This is especially visible for larger libraries whose tests interact with global state, filesystem state, module loading, or timeouts. For example, `graceful-fs` consistently shows unstable behaviour across runs.

The small number of repeated runs we fit in our budget (4 runs at temperature 1, 1 total temperature 0 run) are not enough to mitigate this threat. We can reveal large effects and obvious failures, but do not have enough evidence for strong statistical claims about smaller effects.

Threats to Internal Validity

One threat to internal validity that was controlled for is drift in the agent and harness code. Data was collected over several weeks, during which the system itself still evolved. We mitigate this risk by explicitly dividing our results into the ones that were only used to support architectural decisions (Section 5.2.1), and the ones used to answer the RQs. The latter are all restricted to a single agent commit.

The residual phase also introduces a possible attribution threat. The intended interpretation is that residual feedback adds tests for survivor clusters not already targeted by the public-API item loop. However, after residual items have been generated, the repair phase may edit existing generated files as well. The final post-residual score could therefore not always be the score of the original public API suite plus only newly added residual tests. It is the score of the suite after residual generation and any repairs the agent deemed necessary. The residual effect should be interpreted as the effect of the residual phase as a whole, not just as the effect of adding residual test files.

Finally, there is a threat that the proprietary GPT-5.4 model might have been modified throughout, or after our experiments. This might limit the replicability and validity of our results.

Threats to Construct Validity

The main quality metric in this thesis is mutation score. Mutation score is useful because it measures whether generated tests detect behavioural changes, and it is stronger than coverage alone. However, mutation score is still only a proxy for real fault-detection ability. Any reported increases in mutation score should be interpreted as evidence of stronger behavioural testing, not as a direct measurement of real-world bug detection.

Coverage metrics have a related limitation. Coverage shows which code was executed, but it does not show whether the generated tests asserted the right behaviour. This is why

the agent can be closer to TestPilot on coverage than on mutation score. Therefore, we interpret coverage as a supporting metric rather than as the main indicator of test quality.

Threats to External Validity

The experiments use the JavaScript library benchmark derived from the original TestPilot study. This makes the comparison with TestPilot meaningful, but it limits generalization. The benchmark consists of small to medium-sized libraries with public APIs that can be explored and called from generated unit tests. We cannot say if the results will hold for applications, where important behaviour is mostly internal and not available through exported APIs.

The system is also tied to the JavaScript/TypeScript ecosystem. Other languages have different testing frameworks, build systems, dependency mechanisms, and mutation-testing tools. The general architecture of an MCP-based tool-enabled agent could still be useful in other ecosystems, but we cannot make any claims about the resulting test effectiveness and cost profile.

Another limitation is our use of a single frontier language model. All main experiments use GPT-5.4 due to budget constraints. This keeps the comparison simple, but it also means the results may not generalize to other models. A different model might benefit more or less from the feedback provided to our agent.

7.3 Future Work

The contributions and limitations of this thesis leave plenty of room for future work. We provide an overview of some possibilities:

- *More experiments* Mitigating our threat to conclusion validity would require more experiments under the same conditions. Additionally, there are many possible conditions that have been left unevaluated due to budget and time constraints. Different LLMs at different LLM temperatures would be interesting first steps. Our selection of 1 LLM sampled at 2 temperatures is unlikely to coincidentally be the most cost efficient and/or effective solution, so more research here is recommended.

- *More mutation testing on the same results* As mentioned in Background Section 2.3.3, mutation scores on a fixed target are not deterministic. Re-running mutation analysis multiple times would give more insight in the variance of the results. This would allow for drawing more fine-grained conclusions on existing data. For example, if the variance

turned out very low, the small temperature differences between the temperature 0 and temperature 1 experiments (Figure 6.5) become more interesting. A prerequisite of this would be to run more experiments in total, to first mitigate the non-determinism inherent in the agentic test generation process.

- *Integrating Stryker MCP with state-of-the-art coding agents.* Building a custom agent let us run controlled, transparent experiments in which every tool call, budget, and feedback path was observable. This meant we had to sacrifice the mature code context management, repository indexing, project exploration, and conversation compaction that production coding agents such as OpenCode [57], Codex [46], or Claude Code [2] already provide. A natural next step is to attach the Stryker MCP server and a mutation-feedback prompt to such a harness and measure whether their stronger context management combined with the same feedback signal creates better results.

- *Fine-grained test rollback* As discussed earlier in this chapter and in Section 6.5.1 of Experimental Results, we currently have no way of recovering tests that only fail after attempted mutation feedback integration. The results could add nuance to our observation that the effects of fine-grained mutation feedback are, at best, equally useful and confusing.

- *More ablation studies* Due to the limited budget of the thesis, multiple design decisions were made after a pilot run, and only systematically evaluated together. Ablation studies on the API item loop or the inclusion of `ripgrep` and other MCP tools would give insight into which capabilities of the agent are the most efficient upgrades token wise.

- *Parameter tuning* The test generation loop contains some parameters that have been chosen on anecdotal observations during development, because emperically evaluating their optimal values would exceed the budget. For example, studying different tool call budgets (Section 5.5.1) could help optimize the tradeoff between successfully fixing bugs after many retries and wasting tokens in an unproductive tool call loop. Likewise, allowing more retries during the suite-level repair phases (Section 5.5.3) could prove worthwhile if it helps prevent failures in a relatively large amount of projects.

- *Real bug detection* Bug detection is a powerful metric that we only approximate with mutation score. Future work could adapt the agent for experiments on datasets with known bugs. For example, the BugsJS dataset contains large JavaScript applications and libraries with known bugs [28]. However, it is likely that the BugsJS data has been

included as training data for modern LLMs, contaminating the results and requiring a mitigation strategy.

- *Integrating other test quality signals inside the feedback loop* We used mutation analysis as the only test quality signal inside the test generation loop. Perhaps coverage reports could complement or replace the mutation feedback signal. Other interesting signals would be test flakiness checks or test runtime budgets. Studying how these signals can complement or replace each other inside this agentic environment is an open research direction.

7.4 Practical Applicability

As discussed earlier in Section 5.2, the agent's design changed dramatically to target some failure modes that appeared during a pilot run. Our simple agent consisting only of filesystem exploration tools plus Stryker, produced low mutation scores or failed to produce any at all. The introduction of programmatic API exploration, repair and residual phases improved the performance dramatically, but reduced the versatility and autonomy of our agent.

The API exploration phase is designed to work with libraries which expose an API that contains useful test targets. This makes our approach effective for the utility libraries in our benchmark, but it is likely less suitable for other classes of systems. Applications with user interfaces, databases, network dependencies, and other complexities would likely need a modified version of the harness.

In practice, we see the best use case of the project as a test generation augmentation tool for library developers. If a developer finds themselves with an untested library, a run of the agent might provide a useful starting point. If not, a developer could augment their current test suite by manually selecting strong tests from the system's output.

For other applications, we recommend experimenting with the utilization of the Stryker MCP server as a standalone augmentation for an existing coding harness (also recommended as future work in the previous section). Modern coding agents are likely to drastically outperform our simple agent in project exploration and context management. With the right prompt, an agent could autonomously utilize Stryker MCP to target and fix weak points in the test suite. This is a promising direction since it is essentially a more versatile version of our 'residual feedback' phase, which has already proven useful in experiments.

7.5 Conclusion

This thesis investigates whether an MCP-based, feedback-driven agent can generate effective JavaScript unit tests, and whether StrykerJS mutation feedback improves the quality of those tests. To answer these questions, we built a research harness that provides an agent with tools for filesystem access, code search, test execution, API exploration, and mutation testing. The system was evaluated on a JavaScript library benchmark derived from the TestPilot study and compared against GPT-5.4 replications of TestPilot's light-context generation approach.

The agentic system performs well on the benchmark. Compared with GPT-5.4 TestPilot, it achieves higher mutation scores on most libraries while producing far fewer passing tests. This suggests that the agent tends to more targeted tests. The agent also improves suite-level robustness by validating generated tests both at item level and at full-suite level, although it does not eliminate all failure modes.

The effect of mutation feedback is less clear. Ablating per-item mutation feedback shows no measurable effect on mutation score. However, mutation analysis is still useful when it is targeted differently. The residual feedback phase consistently improves mutation score by using project-wide mutation results. These are used to identify remaining weakly tested source regions and convert them into additional work items, which results in a substantial improvement for a small token overhead.

We find that mutation analysis is more effective in this system as a suite-level targeting mechanism than as direct feedback scoped on the unit test level. Combined with the improvements over minimal context strategy used by TestPilot, we conclude that there is potential for traditional software-engineering tools like mutation analysis in agent-based test generation.

Data and Code Availability

The agentic test generator and benchmark harness (agentic-stryker) is publicly available at `https://github.com/eefscheef/agentic-stryker`. All experiments in this thesis were performed at development-history commit `63be77f`. The complete run data is archived at `https://doi.org/10.5281/zenodo.21326300`. The analysis pipeline that generates every table and figure in this thesis from that data is available at `https://github.com/eefscheef/agentic-stryker-tables`.

The original TestPilot data (Codex cushman, GPT-3.5-turbo, StarCoder) is available in the artifact by Schäfer et al. [55] at `https://doi.org/10.6084/m9.figshare.23653371`.

The GPT-5.4 TestPilot replications were produced with our fork of TestPilot, available at `https://github.com/eefscheef/testpilot2`, and the retroactive mutation analysis that added Stryker mutation reports to all TestPilot runs is available at `https://github.com/eefscheef/mutation-pilot`.

BIBLIOGRAPHY

[1] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" en, in *Proceedings of the 27th international conference on Software engineering - ICSE '05*, St. Louis, MO, USA: ACM Press, 2005, p. 402. DOI: 10.1145/1062455.1062530.

[2] Anthropic. "Claude Code," Accessed: Jun. 17, 2026. [Online]. Available: https://www.anthropic.com/claude-code.

[3] Anthropic. "Pricing." Claude API documentation, Accessed: May 26, 2026. [Online]. Available: https://platform.claude.com/docs/en/about-claude/pricing.

[4] E. Arteca, S. Harner, M. Pradel, and F. Tip, "Nessie: Automatically testing JavaScript APIs with asynchronous callbacks," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22, New York, NY, USA: Association for Computing Machinery, Jul. 2022, pp. 1494–1505. DOI: 10.1145/3510003.3510106.

[5] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin, "A Neural Probabilistic Language Model," *Journal of Machine Learning Research*, vol. 3, no. Feb, pp. 1137–1155, 2003.

[6] M. S. Bouafif, M. Hamdaqa, and E. Zulkoski, *PRIMG : Efficient LLM-driven Test Generation Using Mutant Prioritization*, arXiv:2505.05584 [cs], May 2025. DOI: 10.48550/arXiv.2505.05584.

[7] T. Brown et al., "Language Models are Few-Shot Learners," in *Advances in Neural Information Processing Systems*, vol. 33, Curran Associates, Inc., 2020, pp. 1877–1901.

[8] D. Cai, B. Cao, W. Lam, Z. Zhang, and Y. Zou, "On the Worst Prompt Performance of Large Language Models," in *Advances in Neural Information Processing Systems 37*, Vancouver, BC, Canada: Neural Information Processing Systems Foundation, Inc. (NeurIPS), 2024, pp. 69 022–69 042. DOI: 10.52202/079017-2205.

[9] A. Celik and Q. H. Mahmoud, "A Review of Large Language Models for Automated Test Case Generation," en, *Machine Learning and Knowledge Extraction*, vol. 7, no. 3, p. 97, Sep. 2025, Publisher: Multidisciplinary Digital Publishing Institute. DOI: 10.3390/make7030097.

[10] M. Chen et al., *Evaluating Large Language Models Trained on Code*, arXiv:2107.03374 [cs], Jul. 2021. DOI: 10.48550/arXiv.2107.03374.

[11] Y. Chen, Z. Hu, C. Zhi, J. Han, S. Deng, and J. Yin, *ChatUniTest: A Framework for LLM-Based Test Generation*, arXiv:2305.04764 [cs], May 2024. DOI: 10.48550/arXiv.2305.04764.

[12] M. Collina. "mcp-ripgrep: An MCP server to wrap ripgrep." version 0.4.0, Accessed: Jun. 23, 2026. [Online]. Available: https://github.com/mcollina/mcp-ripgrep.

[13] B. Cottier, B. Snodin, D. Owen, and T. Adamczewski, *LLM inference prices have fallen rapidly but unequally across tasks*, Epoch AI Data Insight, Mar. 2025.

[14] E. Daka and G. Fraser, "A Survey on Unit Testing Practices and Problems," in *2014 IEEE 25th International Symposium on Software Reliability Engineering*, Naples, Italy: IEEE, Nov. 2014, pp. 201–211. DOI: 10.1109/ISSRE.2014.11.

[15] A. M. Dakhel, A. Nikanjam, V. Majdinasab, F. Khomh, and M. C. Desmarais, "Effective test generation using pre-trained Large Language Models and mutation testing," *Information and Software Technology*, vol. 171, p. 107 468, Jul. 2024. DOI: 10.1016/j.infsof.2024.107468.

[16] X. Deng et al., *SWE-Bench Pro: Can AI Agents Solve Long-Horizon Software Engineering Tasks?* arXiv:2509.16941 [cs.SE], Nov. 2025. DOI: 10.48550/arXiv.2509.16941.

[17] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*, arXiv:1810.04805 [cs], May 2019. DOI: 10.48550/arXiv.1810.04805.

[18] J. Du et al., "DependEval: Benchmarking LLMs for Repository Dependency Understanding," in *Findings of the Association for Computational Linguistics: ACL 2025*, W. Che, J. Nabende, E. Shutova, and M. T. Pilehvar, Eds., Vienna, Austria: Association for Computational Linguistics, Jul. 2025, pp. 7150–7179. DOI: 10.18653/v1/2025.findings-acl.373.

[19] C. Foster et al., *Mutation-Guided LLM-based Test Generation at Meta*, arXiv:2501.12862 [cs], Jan. 2025. DOI: 10.48550/arXiv.2501.12862.

[20] G. Fraser and A. Arcuri, "EvoSuite: Automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ACM, 2011, pp. 416–419. DOI: 10.1145/2025113.2025179.

[21] N. Friedman. "Introducing github copilot: Your ai pair programmer." GitHub Blog, Accessed: Jun. 10, 2026. [Online]. Available: https://github.blog/news-insights/product-news/introducing-github-copilot-ai-pair-programmer/.

[22] "Function calling - OpenAI API." en-US, Accessed: Oct. 22, 2025. [Online]. Available: https://platform.openai.com.

[23] garypretty. "Add tools to custom agents - Microsoft Copilot Studio." en-us, Accessed: Oct. 22, 2025. [Online]. Available: https://learn.microsoft.com/en-us/microsoft-copilot-studio/advanced-plugin-actions.

[24] GitHub. "Github copilot is now available to individual developers." GitHub Changelog, Accessed: Jun. 10, 2026. [Online]. Available: https://github.blog/changelog/2022-06-21-github-copilot-is-now-available-to-individual-developers/.

[25] T. Godage, Sivaraj Nimishan, S. Vasanthapriyan, V. Palanisamy, C. Joseph, and S. Thuseethan, "Evaluating the Effectiveness of Large Language Models in Automated Unit Test Generation," in *2025 5th International Conference on Advanced Research in Computing (ICARC)*, Feb. 2025, pp. 1–6. DOI: 10.1109/ICARC64760.2025.10962997.

[26] Google. "Gemini developer api pricing." Google AI for Developers documentation, Accessed: May 26, 2026. [Online]. Available: https://ai.google.dev/gemini-api/docs/pricing.

[27] S. Gu, N. Nashid, and A. Mesbah, *LLM Test Generation via Iterative Hybrid Program Analysis*, Version Number: 2, 2025. DOI: 10.48550/ARXIV.2503.13580.

[28] P. Gyimesi et al., "BugsJS: A Benchmark of JavaScript Bugs," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, ISSN: 2159-4848, Apr. 2019, pp. 90–101. DOI: 10.1109/ICST.2019.00019.

[29] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *2012 34th International Conference on Software Engineering (ICSE)*, ISSN: 1558-1225, Jun. 2012, pp. 837–847. DOI: 10.1109/ICSE.2012.6227135.

[30] X. Hou, Y. Zhao, S. Wang, and H. Wang, *Model Context Protocol (MCP): Landscape, Security Threats, and Future Research Directions*, arXiv:2503.23278 [cs], Oct. 2025. DOI: 10.48550/arXiv.2503.23278.

[31] W. Howden, "Weak Mutation Testing and Completeness of Test Sets," *IEEE Transactions on Software Engineering*, vol. SE-8, no. 4, pp. 371–379, Jul. 1982. DOI: 10.1109/TSE.1982.235571.

[32] "Introducing the Model Context Protocol \ Anthropic," Accessed: Oct. 18, 2025. [Online]. Available: https://www.anthropic.com/news/model-context-protocol.

[33] C. E. Jimenez et al., "SWE-bench: Can language models resolve real-world GitHub issues?" In *Proceedings of the International Conference on Learning Representations*, 2024.

[34] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, *Large Language Models are Zero-Shot Reasoners*, Version Number: 4, 2022. DOI: 10.48550/ARXIV.2205.11916.

[35] B. Kushigian, S. J. Kaufman, R. Featherman, H. Potter, A. Madadi, and R. Just, "Equivalent mutants in the wild: Identifying and efficiently suppressing equivalent mutants for java programs," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ACM, 2024, pp. 654–665. DOI: 10.1145/3650212.3680310.

[36] J. Li et al., *The Tool Decathlon: Benchmarking Language Agents for Diverse, Realistic, and Long-Horizon Task Execution*, arXiv:2510.25726 [cs.CL], Feb. 2026. DOI: 10.48550/arXiv.2510.25726.

[37] X. Li, "Comparative analysis and prospect of RNN and Transformer," en, *Applied and Computational Engineering*, vol. 75, pp. 178–184, Jul. 2024. DOI: 10.54254/2755-2721/75/20240535.

[38] Y. Li, "A Practical Survey on Zero-shot Prompt Design for In-context Learning," in *Proceedings of the Conference Recent Advances in Natural Language Processing - Large Language Models for Natural Language Processings*, arXiv:2309.13205 [cs], 2023, pp. 641–647. DOI: 10.26615/978-954-452-092-2_069.

[39] R. Lipton, "Fault diagnosis of computer programs. student report," *Carnegie Mellon University*, vol. 2, p. 2, 1971.

[40] S. Lukasczyk and G. Fraser, "Pynguin: Automated unit test generation for python," in *Proceedings of the 44th International Conference on Software Engineering: Companion Proceedings*, ACM, 2022, pp. 168–172. DOI: 10.1145/3510454.3516829.

[41] Microsoft. "Fiscal year 2024 second quarter earnings conference call." Microsoft Investor Relations, Accessed: Jun. 10, 2026. [Online]. Available: https://www.microsoft.com/en-us/investor/events/fy-2024/earnings-fy-2024-q2.

[42] Microsoft. "Mutation testing." Microsoft Learn documentation, Accessed: Jun. 10, 2026. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/core/testing/mutation-testing.

[43] Model Context Protocol. "Filesystem MCP Server." Reference implementation of a Model Context Protocol server for filesystem operations. Accessed 23 June 2026, Accessed: Jun. 23, 2026.

[44] "Official page for Language Server Protocol," Accessed: Oct. 22, 2025. [Online]. Available: https://microsoft.github.io/language-server-protocol/.

[45] OpenAI, *Introducing GPT-4.5*, OpenAI, Feb. 2025.

[46] OpenAI. "OpenAI Codex," Accessed: Jun. 17, 2026. [Online]. Available: https://openai.com/codex/.

[47] OpenAI. "Pricing." OpenAI API documentation, Accessed: May 26, 2026. [Online]. Available: https://developers.openai.com/api/docs/pricing.

[48] T. Patwardhan et al., *GDPval: Evaluating AI Model Performance on Real-World Economically Valuable Tasks*, arXiv:2510.04374 [cs.LG], Oct. 2025. DOI: 10.48550/arXiv.2510.04374.

[49] PIT Mutation Testing. "PIT mutation testing." Official documentation, Accessed: Jun. 10, 2026. [Online]. Available: https://pitest.org/.

[50] Y. Qin et al., "ToolLLM: Facilitating large language models to master 16000+ real-world APIs," in *Proceedings of the International Conference on Learning Representations*, 2024.

[51] R. Ravi, D. Bradshaw, S. Ruberto, G. Jahangirova, and V. Terragni, "LLMLOOP: Improving LLM-Generated Code and Tests through Automated Iterative Feedback Loops," en,

[52] L. Reynolds and K. McDonell, *Prompt Programming for Large Language Models: Beyond the Few-Shot Paradigm*, arXiv:2102.07350 [cs], Feb. 2021. DOI: 10.48550/arXiv.2102.07350.

[53] S. Roy Chowdhury et al., "Static Program Analysis Guided LLM Based Unit Test Generation," in *Proceedings of the 8th International Conference on Data Science and Management of Data (12th ACM IKDD CODS and 30th COMAD)*, ser. CODS-COMAD '24, New York, NY, USA: Association for Computing Machinery, Jun. 2025, pp. 279–283. DOI: 10.1145/3703323.3703742.

[54] R. Sapkota, K. I. Roumeliotis, and M. Karkee, *AI Agents vs. Agentic AI: A Conceptual Taxonomy, Applications and Challenges*, Version Number: 5, 2025. DOI: 10.48550/ARXIV.2505.10468.

[55] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, *An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation*, arXiv:2302.06527 [cs], Dec. 2023. DOI: 10.48550/arXiv.2302.06527.

[56] A. See, P. J. Liu, and C. D. Manning, *Get To The Point: Summarization with Pointer-Generator Networks*, arXiv:1704.04368 [cs], Apr. 2017. DOI: 10.48550/arXiv.1704.04368.

[57] SST. "OpenCode: Ai coding agent for the terminal," Accessed: Jun. 17, 2026. [Online]. Available: https://opencode.ai.

[58] Stanford Institute for Human-Centered Artificial Intelligence, "The 2025 ai index report," Stanford University, Tech. Rep., 2025.

[59] "Stryker Mutator," Accessed: Oct. 8, 2025. [Online]. Available: https://stryker-mutator.io/.

[60] Stryker Mutator. "Mutation server protocol." GitHub repository, Accessed: Jun. 11, 2026. [Online]. Available: https://github.com/stryker-mutator/editor-plugins/tree/main/packages/mutation-server-protocol.

[61] Z. Tian, H. Shu, D. Wang, X. Cao, Y. Kamei, and J. Chen, "Large Language Models for Equivalent Mutant Detection: How Far Are We?" In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2024, New York, NY, USA: Association for Computing Machinery, Sep. 2024, pp. 1733–1745. DOI: 10.1145/3650212.3680395.

[62] F. Tip, J. Bell, and M. Schäfer, "LLMorpheus: Mutation Testing Using Large Language Models," *IEEE Transactions on Software Engineering*, vol. 51, no. 6, pp. 1645–1665, Jun. 2025. DOI: 10.1109/TSE.2025.3562025.

[63] A. Vaswani et al., "Attention is All you Need," in *Advances in Neural Information Processing Systems*, vol. 30, Curran Associates, Inc., 2017.

[64] G. Wang, Q. Xu, L. C. Briand, and K. Liu, *Mutation-Guided Unit Test Generation with a Large Language Model*, arXiv:2506.02954 [cs], Aug. 2025. DOI: 10.48550/arXiv.2506.02954.

[65] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software Testing With Large Language Models: Survey, Landscape, and Vision," *IEEE Transactions on Software Engineering*, vol. 50, no. 4, pp. 911–936, Apr. 2024. DOI: 10.1109/TSE.2024.3368208.

[66] Z. Wang, Z. Chu, T. V. Doan, S. Ni, M. Yang, and W. Zhang, "History, development, and principles of large language models: An introductory survey," en, *AI and Ethics*, vol. 5, no. 3, pp. 1955–1971, Jun. 2025. DOI: 10.1007/s43681-024-00583-7.

[67] J. Wei et al., "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models," in *Advances in Neural Information Processing Systems*, vol. 35, Curran Associates, Inc., 2022, pp. 24 824–24 837.

[68] C. White et al., "Livebench: A challenging, contamination-free llm benchmark," *arXiv preprint arXiv:2406.19314*, vol. 4, 2024.

[69] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Berlin, Heidelberg: Springer, 2012. DOI: 10.1007/978-3-642-29044-2.

[70] C. Yang, J. Chen, B. Lin, J. Zhou, and Z. Wang, *Enhancing LLM-based Test Generation for Hard-to-Cover Branches via Program Analysis*, arXiv:2404.04966 [cs] version: 1, Apr. 2024. DOI: 10.48550/arXiv.2404.04966.

[71] Q. Zhang, C. Fang, S. Gu, Y. Shang, Z. Chen, and L. Xiao, *Large Language Models for Unit Testing: A Systematic Literature Review*, arXiv:2506.15227 [cs], Jun. 2025. DOI: 10.48550/arXiv.2506.15227.

APPENDIX A: PILOT AGENT RUN DATA

We provide some preliminary results achieved by a primitive version of the agent. These results are used as motivation for some decisions made in the final architecture in the Design Rationale (Section 5.2): Table 5.2 in that section summarises the observations, and this appendix collects the per-library data behind them. This data should only be viewed in that context, and not as a full controlled experiment. This version of the agent was not a clean subset of the final agent (single conversation per library, no code search, no test execution outside Stryker), and predates some targeted fixes causing library-specific failures.

A.1 Pilot Configuration and Tool Surface

The pilot used a simpler test generation process than the current one described in Chapter 5: For each library, the agent was prompted once to inspect the package, write generated Mocha tests using only Node built-ins, and then run a baseline mutation test. The model could then use the returned surviving mutants as feedback, before requesting one final validation mutation run. The agent was limited to at most two successful `strykerMutationTest` calls, but only on the prompt level. The first successful call was meant to be the baseline and feedback, and the second as validation.

This version only had access to the filesystem and Stryker MCP servers. These tool calls were recorded:

- **Stryker MCP tools:**
  - `strykerMutationTest`
  - `strykerMutantDetails`
- **Filesystem MCP tools:**
  - `create_directory`
  - `edit_file`
  - `list_allowed_directories`
  - `list_directory`
  - `read_file/read_text_file`
  - `read_multiple_files`

- `search_files`
- `write_file`

There was no ripgrep MCP server in this pilot. The `search_files` tool was the generic filesystem MCP search tool, not targeted content search. The pilot also had no `runTests`, `exploreAPI`, `getFunctionBody`, `read_file_range`, verdicts, or small mutation scopes enforced on the harness side.

The `strykerMutationTest` tool did already support targeted modes such as `'files'`, `'survivors'`, and `'mutants'`, but the prompt did not contain any specific instructions. Besides, the 2-tool call limit made it sensible to just apply the default `'all'` mode, which is what the agent did for practically every run.

A.2 Mutation-Score Comparison

Table A.1 lists the per-library mutation scores for GPT-Turbo, the two GPT-5.4 TestPilot budgets, and the pilot agent. We attempted test generation for a set of 20 libraries (fs-extra was marked incompatible due to an incompatibility that was later fixed). Only 12 libraries produced a pilot mutation score; the other eight generated test files (this version of the agent produced one test file per library) failed the harness' independent Mocha evaluation and are counted as 0.0%. These are the failure counts cited in Section 5.2.1 and reported in Section A.5.

Table A.1: Mutation score comparison across GPT-Turbo, the two GPT-5.4 TestPilot variants, and the GPT-5.4 agent. Δ reports the absolute per-project difference in percentage points relative to GPT-Turbo. No-passing-test failures are counted as 0.0% mutation score. Row groups follow the original benchmark grouping: Nessie packages, additional GitHub packages, and GitLab packages.

Project	**GPT-Turbo**	**GPT-5.4-500**		**GPT-5.4-1000**		**GPT-5.4-agent**	
	Mut. Score	**Mut. Score**	**Δ (pp)**	**Mut. Score**	**Δ (pp)**	**Mut. Score**	**Δ (pp)**
glob	22.4%	33.9%	**+11.5**	44.4%	**+22.0**	0.0%	-22.4
graceful-fs	59.7%	26.7%	-33.0	27.0%	-32.7	0.0%	-59.7
jsonfile	13.1%	68.9%	**+55.7**	55.7%	**+42.6**	0.0%	-13.1
q	38.7%	55.3%	**+16.6**	54.4%	**+15.7**	33.6%	-5.1
node-dir	26.6%	60.9%	**+34.3**	67.1%	**+40.6**	56.2%	**+29.6**
zip-a-folder	8.3%	95.8%	**+87.5**	95.8%	**+87.5**	100.0%	**+91.7**
quill-delta	45.8%	76.9%	**+31.1**	78.3%	**+32.5**	0.0%	-45.8
complex.js	34.6%	63.9%	**+29.3**	61.1%	**+26.5**	64.2%	**+29.6**
pull-stream	45.6%	69.8%	**+24.3**	73.4%	**+27.8**	0.0%	-45.6
countries-and-timezones	5.2%	16.4%	**+11.2**	16.6%	**+11.5**	0.0%	-5.2
simple-statistics	47.0%	76.6%	**+29.7**	76.7%	**+29.7**	32.1%	-14.8
plural	22.5%	26.4%	**+3.9**	34.3%	**+11.8**	91.6%	**+69.1**
dirty	41.8%	69.4%	**+27.5**	69.4%	**+27.5**	76.7%	**+34.9**
geo-point	56.5%	74.4%	**+17.8**	80.5%	**+24.0**	0.0%	-56.5
uneval	54.2%	89.8%	**+35.6**	89.8%	**+35.6**	91.5%	**+37.3**
image-downloader	10.0%	0.0%	-10.0	51.2%	**+41.2**	97.7%	**+87.7**
crawler-url-parser	22.1%	64.2%	**+42.0**	68.6%	**+46.5**	74.8%	**+52.7**
gitlab-js	7.8%	9.5%	**+1.7**	4.7%	-3.0	0.0%	-7.8
core	35.0%	51.5%	**+16.5**	58.4%	**+23.3**	78.2%	**+43.1**
omnitool	50.7%	57.1%	**+6.4**	58.7%	**+8.0**	26.3%	-24.4
Median score	34.8%	62.4%	–	59.9%	–	32.9%	–
Median Δ	–	–	**+21.0**	–	**+27.0**	–	-5.1

A.3 Pilot Agent Coverage and Mutation Scores

Table A.2 shows the pilot agent's results in detail: The number of passing generated tests and the statement coverage, branch coverage, and mutation score per library. Coverage cells are empty for the eight libraries whose generated tests did not pass the independent Mocha evaluation, discussed in Section A.5.

Table A.2: Statement and branch coverage and mutation score for the GPT-5.4-agent run.

Project	**Passing Tests**	**Stmt Cov**	**Branch Cov**	**Mutation Score**
glob	0	–	–	–
graceful-fs	0	–	–	–
jsonfile	0	–	–	–
q	1	68.8%	59.6%	33.6%
node-dir	1	80.3%	72.7%	56.2%
zip-a-folder	1	100.0%	100.0%	100.0%
quill-delta	0	–	–	–
complex.js	1	87.8%	68.2%	64.2%
pull-stream	0	–	–	–
countries-and-timezones	0	–	–	–
simple-statistics	1	51.8%	39.4%	32.1%
plural	1	100.0%	90.9%	91.6%
dirty	1	92.5%	86.5%	76.7%
geo-point	0	–	–	–
uneval	1	100.0%	100.0%	91.5%
image-downloader	1	100.0%	93.8%	97.7%
crawler-url-parser	1	96.4%	88.8%	74.8%
gitlab-js	0	–	–	–
core	1	98.6%	88.6%	78.2%
omnitool	1	76.7%	69.9%	26.3%

A.4 Token Usage

Table A.3 lists, per library, the lines of code, exported functions under test (FUT), and input and output token totals for the two GPT-5.4 TestPilot budgets and the pilot agent. We can see the context growth example from Section 5.2.2: on `uneval`, a single early file read was re-sent on every following model call and accounts for roughly 2.45M of the pilot's 2.57M input tokens. The totals also show the inverted token balance: the pilot spends almost all tokens on input, whereas TestPilot spends most on output.

Table A.3: Token usage comparison between GPT-5.4 TestPilot and the GPT-5.4 agent. All values are actual counts from the API.

			GPT-5.4-500		**GPT-5.4-1000**		**GPT-5.4-agent**	
Project	**LoC**	**FUT**	**Input**	**Output**	**Input**	**Output**	**Input**	**Output**
glob	314	21	42k	139k	59k	177k	245k	4k
graceful-fs	208	187	308k	1.129M	340k	1.273M	249k	7k
jsonfile	46	4	11k	41k	8k	34k	294k	18k
q	736	98	224k	773k	426k	738k	443k	5k
node-dir	244	6	48k	145k	54k	173k	261k	9k
zip-a-folder	25	3	4k	18k	5k	20k	89k	5k
quill-delta	395	36	104k	328k	178k	333k	425k	11k
complex.js	393	52	130k	368k	130k	367k	384k	4k
pull-stream	308	24	46k	185k	50k	202k	398k	12k
countries-and-timezones	78	7	21k	57k	23k	60k	458k	2k
simple-statistics	917	89	296k	804k	288k	808k	501k	8k
plural	53	4	9k	25k	10k	25k	192k	6k
dirty	89	34	138k	298k	167k	342k	404k	8k
geo-point	76	19	45k	150k	47k	159k	245k	6k
uneval	31	1	4k	11k	5k	13k	2.571M	3k
image-downloader	32	1	2k	10k	4k	17k	160k	6k
crawler-url-parser	100	3	16k	52k	17k	59k	399k	8k
gitlab-js	205	37	234k	328k	106k	330k	307k	7k
core	136	20	66k	206k	65k	207k	320k	8k
omnitool	1603	270	627k	1.846M	627k	1.885M	621k	14k
Total	5989	916	2.375M	6.913M	2.607M	7.221M	8.964M	152k

A.5 Failure Categorisation

Eight of the 20 libraries produced no passing generated tests. We can divide the failures into 3 categories:

- **Missing import, setup, or incorrectly using API (6 libraries):** `glob`, `graceful-fs`, `pull-stream`, `jsonfile`, `geo-point`, and `gitlab-js`. This is the expected failure mode: the agent can struggle with difficult setup logic or complex functions. These failures are the main motivator for improved project exploration, like the introduction of code search in Section 5.2.3.
- **Mutation feedback timed out (1 library):** `countries-and-timezones`. The package is small in source lines but has relatively many mutants. The whole-project mutation feedback was expensive and hit the MCP timeout. This is the failure category cited in Section 5.2.4, motivating forced scoped mutation testing.
- **Harness bug (1 library):** `quill-delta`. The agent generated a TypeScript test file and wrote it to `delta.generated.spec.ts`. However, the Stryker configuration used in the harness at the time only looked for JavaScript test files: `.js`, `.cjs`, and `.mjs`. This led us to miss a mutation score for a generated test that was possibly perfectly fine. The current agent avoids this by requiring generated tests to use a predetermined filename pattern. See Section 5.3.5.

APPENDIX B: EXAMPLE PER-ITEM PROMPT

Listing B.1 shows an abbreviated prompt for one ordinary API item. Some repeated safety rules are omitted here, but the listing shows the two most important implementation features: a stable instruction prefix and a dynamic item context.

```
You are a careful JavaScript/TypeScript test engineer.
Work on exactly one API element. Kill Stryker mutants by writing
Mocha tests with Node built-ins only.

Rules:
- No third-party assertion or mocking libraries.
- Ignore pre-existing project tests. Write all new tests into
   OUTPUT_DIR.
- Every returned mutant must be handled: kill it with tests or
   conclude it
  is not worth more local iteration.
- IMPORT PATHS: do not guess. If Library notes include
  `public API import path from OUTPUT_DIR:`, use it verbatim.
- Issue exactly one tool call per turn.
- After every edit, call `runTests` with `testsDir: OUTPUT_DIR` and
   the
  active item pattern shown below. Do not call Stryker until it
     passes.

Tools:
- Use ripgrep tools for content search.
- Use `read_file_range` when you already know the relevant lines.
- Use `getFunctionBody({ accessPath })` for sibling implementations.

Stryker:
- Per-item Stryker is only iteration feedback.
- Every mutation call must use `mode: 'files'` with the active item
   source
  path and range shown below.
- Never use `mode: 'all'`, `mode: 'survivors'`, or `mode:
   'mutants'` here.
```

```
26
27 ## Workflow
28 1. Read the active item's signature, source location, and body
      below.
29 2. Write a Mocha test at the expected path.
30 3. Call `runTests` using the active item pattern.
31 4. Call `strykerMutationTest({ mode: 'files', format: 'flat',
32    files: [<active item's source path + range>] })`.
33 5. Refine the tests from the returned mutants, within budget.
34 6. Return the verdict via the schema.
35
36 ## Verdict
37 Return a single structured object matching this schema:
38 {
39   status: "completed" | "skipped",
40   skipReason?: string,
41   notes?: string,
42   libraryNotesAppend?: string
43 }
44
45 ## Active run context
46 - DIR=/tmp/agentic-stryker-suite/repo
47 - OUTPUT_DIR=/tmp/agentic-stryker-suite/repo/test/generated
48 - Targeted runTests call:
49   `runTests({ testsDir: OUTPUT_DIR,
50               pattern: "simple-statistics_mean*.test.js" })`.
51 - Mutation scope: pass
52   { path:
        "/tmp/agentic-stryker-suite/repo/dist/simple-statistics.js",
53     range: { start: { line: 123, column: 1 },
54              end:   { line: 130, column: 2 } } }.
55
56 ## Library notes (shared across items)
57 public API import path from OUTPUT_DIR:
      require("../../dist/simple-statistics.js")
58
59 ## Work queue
60 [x] simple-statistics.sum(x: number[]): number
```

````
[>] simple-statistics.mean(x: number[]): number
[ ] simple-statistics.median(x: number[]): number

## Active item
simple-statistics.mean(x: number[]): number
Source location: dist/simple-statistics.js lines 123:1-130:2
Tier: ts-declarations

### Active body
```js
function mean(x) {
  if (x.length === 0) {
    throw new Error("mean requires at least one data point");
  }
  return sum(x) / x.length;
}
```

## Output and file naming
- Write the test for this item to exactly:
  `OUTPUT_DIR/simple-statistics_mean.test.js`.
- Additional files are attributed to this item only if their name
   starts
  with `simple-statistics_mean_` and ends with `.test.js`.

## Budgets
- Budget A: 20 model-actionable tool calls for this item.
- Budget B: 3 successful `strykerMutationTest` calls for this item.
````

Listing B.1: Abbreviated per-item prompt rendered for an API item

APPENDIX C: LIBRARIES UNDER TEST

Table C.1: Benchmark-library corpus characteristics. LOC and domains follow TestPilot's Table 1; valid-mutant counts come from this thesis' Stryker configuration when a compatible run produced that information.

Library	Domain	LOC	Valid mutants	Benchmark notes
`glob`	file system	314	627	1 file: filesystem glob implementation
`fs-extra`	file system	822	–	excluded: overfitting
`graceful-fs`	file system	208	313	lagacy filesystem wrapper
`jsonfile`	file system	46	61	small JSON file helper
`bluebird`	promises	3.1K	–	excluded: incompatible
`q`	promises	736	1,032	dry-run-repair was developed for this library
`rsvp`	promises	565	–	excluded: incompatible
`memfs`	file system	2.2K	–	excluded: incompatible
`node-dir`	file system	244	493	directory traversal
`zip-a-folder`	file system	25	24	zips folders
`js-sdsl`	data structures	1.5K	–	excluded: incompatible
`quill-delta`	document changes	395	851	TypeScript, rich-text manipulation
`complex.js`	numbers/arithmetic	393	1,295	1 file of arithmetic functinos
`pull-stream`	streams	308	474	callback stream protocol library
`countries-and-timezones`	date/timezones	78	2,662	basically a large table of timezone data
`simple-statistics`	statistics	917	2,054	large statistics API
`plural`	text processing	53	175	global state modification
`dirty`	key-value store	89	160	filesystem KV store
`geo-point`	geographical coordinates	76	186	TypeScript class
`uneval`	serialization	31	59	reverse eval, can be unintuitive
`image-downloader`	image handling	32	43	downloads images

Continued on next page

Table C.1: Benchmark-library corpus characteristics (continued).

Library	Domain	LOC	Valid mutants	Benchmark notes
`crawler-url-parser`	URL parser	100	226	repeatedly proven to be a difficult library for LLMs
`gitlab-js`	API wrapper	205	1,057	GitLab API client
`@spacl/core`	access control	136	293	TypeScript access-control package
`omnitool`	utility library	1.6K	4,995	largest TypeScript library

`complex.js`. `complex.js` is a compact library for doing arithmetic on complex numbers. In the version used by TestPilot (and us), there is one implementation file, `complex.js`, and a minified build and TypeScript declarations. Even though the library is small, the public API exposes many functions and the entire library contains 1,295 mutants.

`@spacl/core`. `@spacl/core` is a library for controlling access to filepaths. The source is TypeScript and contains 293 mutants.

`countries-and-timezones`. `countries-and-timezones` is basically a lookup table for country and timezone data. The source contains bundled data and helper functions for building country and timezone records, but we mutate only the compiled `dist/index.js`. This makes for a small target with a very high mutant density: 78 LOC produce 2,662 mutants. To kill these, tests must contain many assertions matching the large amount of available data.

`crawler-url-parser`. `crawler-url-parser` is a single-file URL parser for crawling. It only exposes `parse`, `extract`, and `gettype`, and is responsible for 266 mutants.

`dirty`. `dirty` is a small key-value store which operates on a JSON file. The API is not very large, but filesystem and asynchronous behaviour can be challenging for LLMs.

`geo-point`. `geo-point` Exports a single TypeScript class `GeoPoint` with functions like validation, distance, destination, and GeoJSON/object conversion. It is a small target with 186 valid mutants.

`gitlab-js`. `@nerdvision/gitlab-js` is a TypeScript GitLab API client for JavaScript and TypeScript. The compiled target contains many public classes and 1,057 valid mutants.

glob. `glob` is a single-file glob implementation for JavaScript. The implementation exposes both synchronous and callback-based APIs, producing 627 valid mutants.

graceful-fs. `graceful-fs` is a drop-in replacement for Node's `fs` module aimed at normalizing behaviour across different environments and increased resilience. In the benchmark runs, this library was extremely slow under Stryker because some mutations can time out, even though the final valid-mutant count is only 313.

image-downloader. `image-downloader` downloads an image from a URL to disk. The target is very small with 32 LOC and 43 mutants, but useful tests are difficult to get right. The existing TestPilot tests usually opt for downloading real external resources, while GPT-5.4 runs are wary of downloading actual data.

jsonfile. `jsonfile` is a helper for reading and writing JSON files in Node. The benchmark mutates only 2 files with 61 mutants.

node-dir. `node-dir` provides asynchronous and synchronous directory traversal and helpers for recursive file reading. A pain point for models generating `node-dir` tests is that they often forget to call the iterator's `next()` method manually for some of the API functions, resulting in many timeouts. GPT-5.4 is noticeably better at this and performs better than the 3 models used in TestPilot's original study.

omnitool. `omnitool` is a TypeScript 'general purpose utility library for JavaScript and TypeScript'. It is the largest target in the suite by LOC and mutant count: 1.6K LOC producing 4,995 valid mutants. Pilot runs repeatedly hit the maximum MCP tool call timeout on full mutation tests, which makes `omnitool` an interesting stress test for timeouts.

plural. `plural` is a small library that pluralizes English worlds based on a rule system. The library provides a base set of rules, together with `addRule`, `monkeyPatch`, and `unmonkeyPatch`. It is only 53 LOC, but monkeyPatch mutates `String.prototype` globally, which can make independently generated tests fail when executed together. Repairing these problems requires isolating the prototype state or restoring it after each test that modifies it.

pull-stream. `pull-stream` is a minimal take on data streams and pipes. It is not filesystem-heavy, but contains some complexity in the callback protocol. The benchmark configuration produced 474 valid mutants.

q. `q` is a promise library for JavaScript, marked as deprecated after the release of JavaScript-native promises.

quill-delta. `quill-delta` implements the JSON Delta format for rich-text documents and their modification (in TypeScript). The library is of medium size at 851 mutants.

simple-statistics. `simple-statistics` is a statistics library with a broad API with many statistical fucntions. In fact, it has one of the largest public API surfaces in the benchmark and contains 2,054 valid mutants.

uneval. `uneval` is an approximation of Node.js/v8 `uneval` behaviour, which turns JavaScript values into source-code strings. The target is very small: 31 LOC and 59 valid mutants, but getting assertions exactly right in one shot can be difficult for complex objects. However assertion failures are often easily repairable with one round of execution feedback.

zip-a-folder. `zip-a-folder` is a callback helper for zipping a folder. The library is only 25 LOC with 24 mutants. Though the behaviour is asynchronous and works with real files, `zip-a-folder` performs well across multiple models, likely because of its API's simple nature.

APPENDIX D: FULL EXPERIMENTAL RESULTS

This appendix contains tables with the full measurements per library used to generate the figures in Chapter 6.

D.1 TestPilot Replication Metrics

Table D.1 shows the exact statement, branch, and line coverage and mutation scores for GPT-Turbo and the two GPT-5.4 TestPilot budgets. These are the values used for Figure 6.1 in Section 6.1.3.

Table D.1: Statement, branch, and line coverage and mutation score per package for GPT-Turbo and the two GPT-5.4 TestPilot variants. These are the exact values used in Figure 6.1. GPT-Turbo values are medians over 10 runs, and each GPT-5.4 value is from a single run. Rows are grouped by the original categories: Nessie packages, extra GitHub packages, and GitLab packages.

Project	**GPT-Turbo**				**GPT-5.4-500**				**GPT-5.4-1000**			
	Stmt	**Br**	**Line**	**Mut**	**Stmt**	**Br**	**Line**	**Mut**	**Stmt**	**Br**	**Line**	**Mut**
glob	71.3	66.3	76.1	22.4	62.8	51.4	66.6	33.9	76.0	67.5	80.9	44.4
graceful-fs	49.3	33.3	50.5	59.7	56.7	41.7	57.7	26.7	54.8	40.9	55.8	27.0
jsonfile	38.3	29.4	39.1	13.1	85.1	73.5	84.8	68.9	76.6	70.6	76.1	55.7
q	70.4	53.7	70.9	38.7	80.5	63.5	80.7	55.3	80.6	63.1	80.8	54.4
node-dir	64.3	50.8	72.5	26.6	78.9	67.3	86.9	60.9	82.7	73.5	90.2	67.1
zip-a-folder	84.0	50.0	84.0	8.3	100.0	100.0	100.0	95.8	100.0	100.0	100.0	95.8
quill-delta	73.0	64.3	72.9	45.8	88.9	83.6	88.9	76.9	91.4	86.4	91.4	78.3
complex.js	70.2	46.5	70.3	34.6	86.0	66.9	86.0	63.9	84.5	66.2	84.5	61.1
pull-stream	69.1	52.8	72.4	45.6	86.0	71.5	88.3	69.8	88.9	74.3	89.9	73.4
countries-and-timezones	93.1	69.1	100.0	5.2	99.0	85.1	100.0	16.4	99.0	87.2	100.0	16.6
simple-statistics	87.8	71.3	87.5	47.0	90.3	87.9	90.2	76.6	91.7	88.2	91.6	76.7
plural	73.8	59.1	90.6	22.5	78.5	68.2	92.5	26.4	81.5	77.3	94.3	34.3
dirty	74.5	65.4	79.8	41.8	85.8	78.8	87.6	69.4	84.9	76.9	87.6	69.4
geo-point	87.8	70.6	86.8	56.5	96.3	88.2	96.0	74.4	100.0	97.0	100.0	80.5
uneval	68.8	58.3	71.0	54.2	100.0	95.8	100.0	89.8	100.0	95.8	100.0	89.8
image-downloader	63.6	50.0	62.5	10.0	0.0	0.0	0.0	0.0	75.8	62.5	75.0	51.2
crawler-url-parser	51.4	35.0	54.0	22.1	94.6	81.2	96.0	64.2	94.6	85.0	96.0	68.6
gitlab-js	51.7	16.5	51.9	7.8	59.6	22.5	60.5	9.5	59.6	20.8	60.5	4.7
core	78.3	50.0	77.2	35.0	88.1	72.7	88.2	51.5	91.6	76.1	91.2	58.4
omnitool	74.2	55.2	73.9	50.7	81.6	73.1	82.5	57.1	83.0	76.2	84.0	58.7
Median	70.8	53.2	72.7	34.8	85.9	72.9	87.9	62.4	84.7	76.2	90.0	59.9

D.2 TestPilot Token Usage

Table D.2 contains the token usage per library. This data is used in Figure 6.2 in Section 6.1.5.

Table D.2: Token usage for GPT-Turbo and GPT-5.4 running TestPilot's generation approach. FUT is the number of exported functions under test. GPT-Turbo values are estimated upper bounds (input tokens estimated as prompt length divided by four. output tokens as the output token limit times number of completions). GPT-5.4 values are actual counts from the API.

			GPT-Turbo (est.)		**GPT-5.4-500** (actual)		**GPT-5.4-1000** (actual)	
Project	**LoC**	**FUT**	**Input**	**Output**	**Input**	**Output**	**Input**	**Output**
glob	314	21	13k	36k	42k	139k	59k	177k
graceful-fs	208	187	61k	194k	308k	1.129M	340k	1.273M
jsonfile	46	4	5k	12k	11k	41k	8k	34k
q	736	98	100k	225k	224k	773k	426k	738k
node-dir	244	6	30k	29k	48k	145k	54k	173k
zip-a-folder	25	3	3k	6k	4k	18k	5k	20k
quill-delta	395	36	74k	102k	104k	328k	178k	333k
complex.js	393	52	48k	155k	130k	368k	130k	367k
pull-stream	308	24	21k	50k	46k	185k	50k	202k
countries-and-timezones	78	7	7k	20k	21k	57k	23k	60k
simple-statistics	917	89	112k	223k	296k	804k	288k	808k
plural	53	4	2k	9k	9k	25k	10k	25k
dirty	89	34	30k	50k	138k	298k	167k	342k
geo-point	76	19	18k	41k	45k	150k	47k	159k
uneval	31	1	2k	4k	4k	11k	5k	13k
image-downloader	32	1	2k	2k	2k	10k	4k	17k
crawler-url-parser	100	3	11k	12k	16k	52k	17k	59k
gitlab-js	205	37	27k	80k	234k	328k	106k	330k
core	136	20	47k	62k	66k	206k	65k	207k
omnitool	1603	270	270k	667k	627k	1.846M	627k	1.885M
Total	5989	916	883k	1.979M	2.375M	6.913M	2.607M	7.221M

D.3 Agent vs. TestPilot Per-Library Metrics

Table D.3 lists the mutation score, statement and branch coverage, and generation cost for the two GPT-5.4 TestPilot budgets and the final agent. This data backs the summary in Table 6.2 and the dumbbell in Figure 6.4 (Section 6.3).

Table D.3: Per-library mutation score, statement and branch coverage, and generation cost for the two GPT-5.4 TestPilot budgets and the final GPT-5.4 agent. *Mut*, *Stmt*, and *Br* are mutation score and statement and branch coverage in percent. *Cost* is the USD generation cost for that library. Failing to generate any passing tests counts 0.0% mutation score. Agent values are means over the two temperature 1 runs, and the agent mutation score is the final (post-residual) score.

Project	**TestPilot-500**				**TestPilot-1000**				**Agent**			
	Mut	**Stmt**	**Br**	**Cost**	**Mut**	**Stmt**	**Br**	**Cost**	**Mut**	**Stmt**	**Br**	**Cost**
glob	33.9	62.8	51.4	$1.45	44.4	76.0	67.5	$1.84	51.8	77.7	69.8	$0.70
graceful-fs	26.7	56.7	41.7	$11.68	27.0	54.8	40.9	$13.15	96.8	60.8	51.5	$7.49
jsonfile	68.9	85.1	73.5	$0.42	55.7	76.6	70.6	$0.35	92.6	98.9	92.6	$0.28
q	55.3	80.5	63.5	$8.01	54.4	80.6	63.1	$7.91	65.9	86.7	75.6	$2.61
node-dir	60.9	78.9	67.3	$1.51	67.1	82.7	73.5	$1.79	72.8	84.1	75.8	$0.63
zip-a-folder	95.8	100.0	100.0	$0.19	95.8	100.0	100.0	$0.21	100.0	100.0	100.0	$0.07
quill-delta	76.9	88.9	83.6	$3.41	78.3	91.4	86.4	$3.55	68.1	75.6	71.3	$1.19
complex.js	63.9	86.0	66.9	$3.85	61.1	84.5	66.2	$3.83	78.5	94.4	80.7	$4.04
pull-stream	69.8	86.0	71.5	$1.91	73.4	88.9	74.3	$2.09	82.7	90.1	82.9	$0.95
countries-and-timezones	16.4	99.0	85.1	$0.59	16.6	99.0	87.2	$0.63	11.4	71.6	55.2	$0.82
simple-statistics	76.6	90.3	87.9	$8.41	76.7	91.7	88.2	$8.44	70.3	84.2	79.4	$2.40
plural	26.4	78.5	68.2	$0.26	34.3	81.5	77.3	$0.27	84.5	98.5	95.5	$0.29
dirty	69.4	85.8	78.8	$3.15	69.4	84.9	76.9	$3.63	56.6	68.4	61.5	$0.34
geo-point	74.4	96.3	88.2	$1.56	80.5	100.0	97.0	$1.65	78.2	96.0	82.8	$0.60
uneval	89.8	100.0	95.8	$0.11	89.8	100.0	95.8	$0.13	0.0	0.0	0.0	$0.07
image-downloader	0.0	0.0	0.0	$0.10	51.2	75.8	62.5	$0.18	96.5	100.0	96.9	$0.18
crawler-url-parser	64.2	94.6	81.2	$0.54	68.6	94.6	85.0	$0.61	29.9	35.6	35.0	$0.20
gitlab-js	9.5	59.6	22.5	$3.57	4.7	59.6	20.8	$3.44	31.0	60.9	37.2	$1.17
core	51.5	88.1	72.7	$2.14	58.4	91.6	76.1	$2.15	82.1	96.5	94.3	$0.66
omnitool	57.1	81.6	73.1	$19.24	58.7	83.0	76.2	$19.63	80.2	91.8	87.4	$7.66
Mean	54.4	79.9	68.7	$3.60	58.3	84.9	74.3	$3.77	66.5	78.6	71.3	$1.62

D.4 Residual-Phase Mutation Scores

Table D.4 lists the exact pre-residual and final mutation scores at both decoding temperatures. This is the data visualised by Figure 6.7 in Section 6.5.2.

Table D.4: Residual feedback effect at both decoding temperatures. *Pre-residual* and *Agent final* are the agent-side `strykerMutationTest` scores before and after the residual phase, and Δ is their difference (Agent final − Pre-residual). Temperature 1 is the mean of two runs and temperature 0 is a single run. No-test failures are scored as 0.0%. Cells marked * were taken from a single run.

	Temperature 1 (mean of two runs)			**Temperature 0**		
Project	**Pre-residual**	**Agent final**	**Δ**	**Pre-residual**	**Agent final**	**Δ**
glob	38.1%	51.8%	+13.6	38.4%	60.6%	+22.2
graceful-fs	97.1%*	96.8%*	−0.3*	86.0%	81.8%	−4.1
jsonfile	82.8%	92.6%	+9.8	65.6%	85.2%	+19.7
q	63.3%	65.9%	+2.7	62.0%	69.1%	+7.0
node-dir	56.5%	72.8%	+16.3	46.6%	64.3%	+17.6
zip-a-folder	100.0%	100.0%	+0.0	100.0%	100.0%	+0.0
quill-delta	64.1%	68.1%	+3.9	65.0%	66.0%	+1.1
complex.js	71.4%	78.5%	+7.1	70.4%	78.3%	+7.9
pull-stream	78.8%	82.7%	+3.9	65.0%	67.9%	+3.0
countries-and-timezones	7.1%	11.4%	+4.3	8.4%	14.1%	+5.6
simple-statistics	67.3%	70.3%	+2.9	67.2%	71.8%	+4.5
plural	34.3%	84.5%	+50.3	18.0%	73.0%	+55.0
dirty	56.6%	56.6%	+0.0	28.1%	28.1%	+0.0
geo-point	73.3%	78.2%	+4.9	84.6%	86.2%	+1.5
uneval	0.0%	0.0%	+0.0	91.5%	91.5%	+0.0
image-downloader	73.3%	96.5%	+23.3	74.4%	86.0%	+11.6
crawler-url-parser	29.9%	29.9%	+0.0	23.9%	36.3%	+12.4
gitlab-js	29.3%	31.0%	+1.8	31.4%	39.5%	+8.1
core	61.4%	82.1%	+20.6	66.5%	87.0%	+20.5
omnitool	78.3%	80.2%	+1.9	76.1%	77.4%	+1.3
Mean	58.1%	66.5%	+8.4	58.5%	68.2%	+9.7
Median	63.7%	75.5%	+3.9	65.3%	72.4%	+6.3