LLMutantKiller: Using Large Language Models to Generate Tests that Kill Mutants

FARIDEH KHALILI, Northeastern University, USA
AIDAN DOMONDON, Northeastern University, USA
HARSHIT GARG, Northeastern University, USA
FRANK TIP, Northeastern University and Amazon Web Services, USA

The primary goal of mutation testing is to assess the quality of an application's test suite. This is accomplished by introducing syntactic changes into a program and determining if any test failures occur for the resulting mutated program, commonly referred to as a *mutant*. If so, the mutant is said to be *killed*, confirming that the test suite is of sufficient quality to detect the introduced fault. A problem arises if a mutant does not impact the behavior of any test. Such a *surviving mutant* may occur for two reasons: either it involves a semantics-preserving program transformation or the test suite is not strong enough. Determining why a mutant survives often involves complex, non-local reasoning. This paper presents an LLM-based test generation technique for killing surviving mutants, implemented in a tool called *LLMutantKiller*. The technique is feedback-directed in the sense that if a test is produced that does not kill a given mutant, the LLM is re-prompted up to a specified number of times with scenario-specific feedback such as syntax errors, dependency violations, or execution logs (e.g., failing assertions) and asked to try again. We evaluate *LLMutantKiller* on 915 randomly selected surviving mutants that were produced by *StrykerJS*, a state-of-the-art mutation testing tool, taken from 13 open-source JavaScript/TypeScript applications. The results show that *LLMutantKiller* kills up to 95.3% of the surviving mutants classified as inducing behavioral changes and that it rarely produces invalid tests.

Additional Key Words and Phrases: Software Testing, Test Generation, Mutation Testing, LLM Integration

ACM Reference Format:
Farideh Khalili, Aidan Domondon, Harshit Garg, and Frank Tip. 2026. *LLMutantKiller*: Using Large Language Models to Generate Tests that Kill Mutants. 1, 1 (July 2026), 23 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Mutation testing [16, 24, 43] is a technique for assessing the quality of a program's tests. It involves introducing a syntactic change into a program and determining if any test failures occur for the resulting mutated program (commonly referred to as a *mutant*). If at least one test failure is observed, the mutant is said to be *killed*. Otherwise, the mutant is said to *survive*, indicating that the test suite may be insufficient. Previous research has demonstrated mutants to have behavioral impacts similar to those of real-world bugs [20, 26, 31], suggesting that uncovering problems through mutation testing prevents real-world bugs later on. There has been strong industrial interest in mutation testing recently, as shown by reports of its use at companies such as Google [44] and Meta [6].

Authors' Contact Information: Farideh Khalili, Northeastern University, USA, khalili.f@northeastern.edu; Aidan Domondon, Northeastern University, USA, domondon.a@northeastern.edu; Harshit Garg, Northeastern University, USA, garg.hars@northeastern.edu; Frank Tip, Northeastern University and Amazon Web Services, USA, f.tip@northeastern.edu.

ACM XXXX-XXXX/2026/7-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Traditionally, mutation testing tools have relied on a rule-based approach for creating mutants in which a set of *mutation operators* describes precisely how to rewrite certain syntactic constructs (e.g., transforming an expression $x + y$ into an expression $x - y$, or removing the body of a function). Such techniques have been implemented in popular tools such as PIT [12, 13], Major [25], and *StrykerJS* [50]. More recently, AI-based techniques have been developed in which a Large Language Model (LLM) is asked how a bug should be introduced into a program [52, 53].

Surviving mutants are one of the main challenges associated with mutation testing. There are two categories of surviving mutants: (i) *equivalent mutants* that reflect semantics-preserving program transformations (e.g., an expression is changed in a way that always computes the same value) and (ii) mutants that survive due to inadequate testing. Surviving mutants are problematic because they make it harder to assess test suite quality and undermine confidence in mutation testing tools. Unfortunately, determining why a mutant survives is difficult, since it often involves complex, non-local reasoning. The problem is significant, as is clear from an abundance of prior work on detecting equivalent mutants using compiler-optimization techniques [29, 37], constraint-based techniques [36, 38], program slicing [23], heuristic techniques [42, 48], and LLM-based techniques [5, 7, 15, 22, 49, 53], and on techniques for test generation to kill surviving mutants (e.g., [8, 18, 32, 54]). Our work complements techniques for detecting equivalent mutants: while determining behavioral equivalence of the original and mutated programs is undecidable in general [24, 35], a single valid mutant-killing test provides a concrete witness that the mutant is not equivalent.

This paper presents a practical LLM-based technique for generating tests that kill surviving mutants. The technique involves prompting an LLM with the original and mutated code and asking it to create a test that exposes behavioral differences, along with guidance for the test generation process. The resulting test is executed against the original code. If it passes, it is then executed against the mutated code. If the test fails on the mutant, several criteria are checked to ensure the solution is satisfactory (e.g., tests must not use third-party libraries or inspect the application's source code). If a test is unsatisfactory (e.g., it fails on the original code, passes on the mutated code, or fails other checks), the LLM is re-prompted with scenario-specific feedback and asked to try again. This process is repeated up to a fixed number of times (10 attempts in our experiments).

We implemented this technique in a tool called *LLMutantKiller*, targeting applications written in JavaScript and TypeScript. In an empirical evaluation, we applied *LLMutantKiller* to 13 subject applications. We generated mutants for these applications using *StrykerJS*, a state-of-the-art mutation testing tool for JavaScript and TypeScript. For these applications, *StrykerJS* generated between 43 and 1,302 mutants, of which between 5 and 927 survived. To support estimating evaluation results over each project's surviving-mutant population, we used Cochran's sample-size formula for proportions with finite-population correction [10] to determine the required number of mutants to sample per project, using a 95% confidence level, a 5% margin of error, and the conservative population-proportion estimate $p = 0.5$. We then randomly sampled the required number of surviving mutants from each project, yielding 915 surviving mutants.

We then used *LLMutantKiller* to generate mutant-killing tests for these surviving mutants using two variations on a standard prompt that includes/excludes the application's existing tests, and ran experiments with 3 LLMs (*claude-sonnet-4.6*, *devstral-2512*, and *llama-3.3-70b-instruct*). In these experiments *LLMutantKiller* produced tests that killed between 218 (*llama-3.3-70b-instruct* without existing tests) and 811 (*claude-sonnet-4.6* without existing tests) surviving mutants.

When interpreting these results, care must be taken because LLMs sometimes produce tests that should be considered invalid, e.g., because they rely on introspection to check whether the original or mutated source code is present. Since LLMs may produce arbitrarily complex code, devising a comprehensive solution to this problem is extremely challenging. To assess the prevalence of the

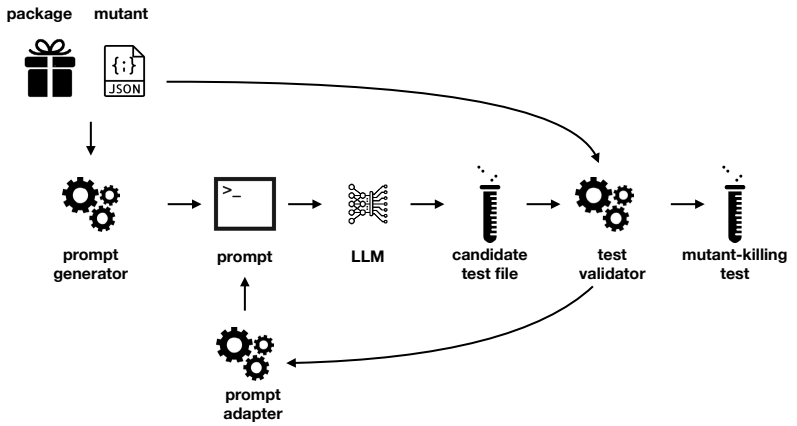

Fig. 1. Overview of approach.

issue, we manually validated all generated tests. This revealed that at most 3.7% of mutants were killed only by invalid tests in any configuration, confirming that invalid tests are not a major factor.

To determine an upper bound on the number of surviving mutants that *could be killed*, we manually classified each of the 915 surviving mutants as either a behavioral change or as an equivalent mutant. This revealed that, of the 915 surviving mutants, 815 were behavioral changes and 100 were equivalent mutants. Putting these results together, we found that *LLMutantKiller* was able to generate valid tests that kill up to 777 (95.3%) of the 815 non-equivalent surviving mutants. To illustrate the potential practical impact on a subject application, extrapolating from the sample of surviving mutants in the *q* library, these tests would increase the project's mutation score from 12.38% to approximately 92.9%.

In summary, this paper makes the following contributions:

- A feedback-directed, LLM-based technique for generating tests that kill surviving mutants.
- A practical tool called *LLMutantKiller* that implements this technique for JavaScript/TypeScript, which will be made available open-source and submitted for artifact evaluation.
- An experimental evaluation on 915 surviving mutants produced by *StrykerJS*, a state-of-the-art mutation testing tool for JavaScript/TypeScript in which *LLMutantKiller* is demonstrated to produce valid tests that kill up to 95.3% of the non-equivalent surviving mutants.

The remainder of this paper is organized as follows. Section 2 presents an overview of our approach. An empirical evaluation of *LLMutantKiller* on 13 subject applications is presented in Section 3. Threats to validity are discussed in Section 4. Related work is presented in Section 5. Finally, conclusions and directions for future work are presented in Section 6.

2 Approach

Our goal is to automatically generate, for each surviving mutant, a single test file (using Jest [1]) that *passes on the original program* and *fails on the mutated version*, thereby killing the mutant. To achieve this, we combine LLM-based test synthesis with an execution-based filtering pipeline that iteratively rejects candidates failing syntactic, dependency, or execution checks.

Figure 1 presents an overview of our approach and shows the main components of *LLMutantKiller*. Our approach takes as primary inputs (i) a JavaScript/TypeScript **package** containing the project under test, and (ii) a JSON description of the **mutant**, including the affected source file and the original and mutated code fragments. These inputs are passed to the **prompt generator**, which constructs an initial **prompt** that is sent to the **LLM**.

(a) System Prompt Template

```
You are a test generation tool that produces unit
      tests designed to kill mutants by exposing
      observable behavioral differences between the
      original and mutated versions of a program.

The generated tests must have the following
      characteristics:
- Tests are deterministic (no randomness, timing
      dependencies, or reliance on the system clock)
- Tests appear natural and idiomatic, as written by
      an experienced human developer
- Tests use the Jest testing framework
- Tests are written in TypeScript
- When importing modules from the project, use
      relative paths only, e.g.:
  import { Something } from
        "{{{importPathToMutatedFile}}}"
- Tests assert only on externally observable
      behavior, not implementation details

The generated tests must NOT:
- Inspect or analyze source code to detect whether a
      mutation is present
- Use absolute file paths
- Use any external libraries except Jest, the
      standard language/runtime, and those explicitly
      mentioned in the prompt
- The use of mocks and spies is permitted, but
      should be used only as a last resort when
      direct observation of behavior is not possible.

FINAL OUTPUT REQUIREMENT
- Return one Jest test file with exactly one
      describe and exactly one test/it. Avoid adding
      extra suites/cases
- The response must consist of only a single fenced
      code block:
```typescript
// <Jest test file containing exactly one test case>
```
Do not include any text, explanations, or comments
      outside the code block.
```

(b) Initial Prompt Template

```
Your task is to write a Jest test case that exposes
      a specific mutation in a JavaScript/TypeScript
      project.


MUTATION DETAILS
The mutation affects the following file:
{{{mutationObj.file}}}
Here is the full content of that file (shown with a
      <PLACEHOLDER> marker):
{{{origFileWithPlaceHolder}}}
In the original code, <PLACEHOLDER> has the value:
{{{mutationObj.originalCode}}}
In the mutated code, <PLACEHOLDER> has the value:
{{{mutationObj.replacement}}}

YOUR TASK
Write one Jest test file that:
    1. Passes when run against the original code.
    2. Fails when run against the mutated code.
    3. Tests the behavior of the mutated file in a
          way that reliably detects the mutation.
The test will be saved at the following path,
      relative to this test-generation project's root:
{{{testCaseRelPath}}}

RESOURCES AVAILABLE
Existing test cases you may use for style or structure
      inspiration, focusing on how they reflect the
      project's expected behavior and typical inputs:
{{{existingTestCases}}}
The test runner configuration (package.json):
{{{packageDotJson}}}

ALLOWED LIBRARIES
You may use:
    1. Any built-in Node.js modules (e.g., fs, path,
          crypto, etc.).
    {{{allowedExternalLibraries}}}


    You may not use any other third-party libraries.
```

Fig. 2. Prompt templates for LLM-driven test generation: (a) system prompt, and (b) the initial prompt. Triple curly braces denote template variables filled by the prompt generator. We evaluate two variants of (b): with or without the RESOURCES AVAILABLE section.

Figure 2 shows templates that are instantiated to obtain (a) the system prompt and (b) the initial prompt used by *LLMutantKiller*, respectively. Here, the template variables (surrounded by triple curly braces) are bound as follows: `importPathToMutatedFile` provides the relative import path from the generated test file to the mutated module and `mutationObj.file`, `mutationObj.originalCode`, `mutationObj.replacement`, and `origFileWithPlaceHolder` specify the mutant to be killed, including the affected file and the original/mutated code fragments, respectively. Following recent work on LLM-based mutation testing [52], we represent mutants by replacing the mutated code region in the source file with `<PLACEHOLDER>` and provide the code fragments needed to instantiate it to produce the original and mutated versions, respectively. Figure 3 shows a concrete example of this

(a) Mutation metadata (excerpt)

```
{
  "file": "util/prop.js",
  "originalCode": ": 'object' === typeof key &&
       'function' === typeof key.exec //regexp",
  "replacement": ": \"\" === typeof key &&
       'function' === typeof key.exec //regexp",
  ...
}
```

(b) Source file with `PLACEHOLDER`

```
module.exports = function prop (key) {
  return key && (
    'string' == typeof key
    ? function (data) { return data[key] }
    <PLACEHOLDER>
    ? function (data) { var v = key.exec(data);
         return v && v[0] }: key
  )}
```

Fig. 3. Concrete example of our mutation representation: (a) mutation metadata, and (b) the affected source file with the mutated region replaced by <PLACEHOLDER>. Example from the `pull-stream` project.

(a) `syntax_error`

```
This test case doesn't meet the criteria (Reminder:
The criteria is that the test case must pass on the
original code,and fail on the mutated code.)

This test case fails with a syntax error with the
following error message:
{{{errorMessage}}}
---
Please try again and give me a new test case.
```

(b) `forbidden_libraries`

```
This test case doesn't meet the criteria (Reminder:
The criteria is that the test case must pass on the
original code,and fail on the mutated code.)

This test case imports these external libraries that
are not allowed for this project:
{{{errorMessage}}}
---
Please try again and give me a new test case.
```

(c) `fail_on_original`

```
This test case doesn't meet the criteria (Reminder:
The criteria is that the test case must pass on the
original code,and fail on the mutated code.)

This test case fails on the original code with the
following log:
{{{errorLog}}}
---
Please try again and give me a new test case.
```

(d) `pass_on_mutation`

```
This test case doesn't meet the criteria (Reminder:
The criteria is that the test case must pass on the
original code,and fail on the mutated code.)

While the test case passes on the original code, it
also passes on the mutated code.

Please try again and give me a new test case.
```

Fig. 4. Prompt adapter feedback prompt templates used for re-prompting, specialized to each failure category.

representation. Furthermore, `testCaseRelPath` is the path where the generated test should be saved and `existingTestCases` and `packageDotJson` denote optional project context, including existing tests and the `package.json` file. In Section 3, we report on experiments with two prompt variants, with and without this optional context section. Finally, `allowedExternalLibraries` specifies external libraries that generated tests may use.

The prompt obtained by instantiating these templates is passed to an LLM, which produces a **candidate test file**. This is then checked by the **test validator**, which first parses the test to check if it is syntactically valid and then checks imported/required modules to ensure that it uses only built-in Node.js modules or project-specific allowed external libraries. In particular, the generated tests are only allowed to use external libraries that are already being used by the subject application or its test suite. If the candidate passes these checks, it is executed on the original program; if the file contains multiple tests and some fail on the original, the validator discards the failing tests and keeps only those that pass. Any remaining tests are then executed on the mutated program and emitted as **mutant-killing tests** if they fail on the mutant.

When a candidate test is produced that does not satisfy all requirements, *LLMutantKiller*'s **prompt adapter** component creates a new prompt. Depending on the type of failure that was observed, this new prompt instantiates one of the templates in Figure 4 with diagnostic information that may include a syntax error message, a list of forbidden dependencies, or an execution log (Figure 4). If querying the LLM with this new prompt yields a new candidate test that again

```
[
  {
    "role": "system",
    "content": "You are a test generation tool that produces unit tests designed to kill mutants ···"
  },
  {
    "role": "user",
    "content": "Your task is to write a Jest test case that exposes a specific mutation ···"
  },
  {
    "role": "assistant",
    "content": "```typescript
    import { GeoPoint } from ···;

    describe('GeoPoint', () => {
      it('should throw an error when creating from an object that is not an object', () => {
          expect(() => GeoPoint.fromObject(null)).toThrowError('Object must have latitude and longitude');
        });
     });```"
  },
  {
    "role": "user",
    "content": "This test case doesn't meet the criteria (Reminder: The criteria is that the test case must
     pass on the original code,and fail on the mutated code.)

    This test case fails on the original code with the following log: (details omitted)
    Please try again and give me a new test case."
  },
  {
    "role": "assistant",
    "content": "```typescript
    import { GeoPoint } from ···;

    describe('GeoPoint', () => {
      it('should throw an error when creating from an object without latitude and longitude', () => {
        expect(() => GeoPoint.fromObject({})).toThrowError('Object must have latitude and longitude');
      });
    });
    ```"
  },
  {
    "role": "user",
    "content": "This test case doesn't meet the criteria (Reminder: The criteria is that the test case must
         pass on the original code,and fail on the mutated code.)
    While the test case passes on the original code, it also passes on the mutated code.
    Please try again and give me a new test case."
  },
  {
    "role": "assistant",
    "content": "```typescript
    import { GeoPoint } from ···;
    describe('GeoPoint', () => {
      it('should throw an error when creating from a non-object', () => {
        expect(() => GeoPoint.fromObject('string')).toThrowError('GeoPoint: Argument must be an object');
      });\n});```"
  }
]
```

Fig. 5. Example scenario produced by *LLMutantKiller* for *node-geo-point* (various details have been removed for space and clarity). Initially a test case is produced that fails on the original code. Following re-prompting, a test case is produced that passes on both the original code and the mutant. Then, after reprompting again a mutant-killing test case is produced.
```

```
{
  "file": "complex.js",
  "originalCode": "var a = this['re'];",
  "replacement": "var a = this[\"\"];",
  ...
}
```

(a)

```
import { Complex } from
      "<repo>/Complex.js/complex.js";

describe("Complex", () => {
  it("should have an asec function that references
       re", () => {
    const originalAsec = Complex.prototype.asec;
    const asecString = originalAsec.toString();
    expect(asecString)
      .toContain("var a = this['re'];");
  });
});
```

(b)

Fig. 6. Example of a behaviorally invalid test for `Complex.js`. The test distinguishes versions by obtaining the source code of the mutated function by invoking `toString` on the function object and asserting on the presence of the original code fragment in the source text.

does not meet the requirements, the LLM is re-prompted up to a specified number of times. In our experiments, we used a maximum of *10 attempts* (i.e., *up to 9 re-prompting attempts* if the initial prompt does not yield a mutant-killing test). The re-prompting process terminates when a mutant-killing test is obtained or when the attempt budget is exhausted.

This approach is implemented in a tool called *LLMutantKiller*. We will release *LLMutantKiller* as open-source software and submit an artifact for evaluation; a preliminary version is included in the supplemental materials associated with this paper. To illustrate our approach, Figure 5 shows a concrete scenario produced by *LLMutantKiller* for the *node-geo-point* subject application. Here, initially a test was produced that fails on the original code. In response to this result, the LLM is re-prompted using the prompt of Figure 4(c), which results in a test that passes on both the original code and the mutant. The LLM was then re-prompted again using the prompt of Figure 4(d), which results in a test that kills the mutant. Each round of re-prompting is modeled by extending the previous chat history, so that the LLM can learn from its previous attempts.

By construction and automated filtering, *LLMutantKiller* only emits tests that (i) are syntactically valid, (ii) respect the dependency constraints, (iii) pass on the original program, and (iv) fail on the mutant. However, a test that satisfies these criteria is not necessarily behaviorally valid. In preliminary experiments, we observed that LLMs may occasionally "cheat" by producing tests that distinguish program versions by exploiting non-behavioral signals (e.g., inspecting source text) rather than by exercising externally observable behavior. Figure 6(a) shows an example of such a scenario in *Complex.js*. Here, *StrykerJS* applied a mutation operator that replaces a string literal with an empty string to change an original code fragment `var a = this['re'];` into `var a = this[\"\"];`. Figure 6(b) shows a mutant-killing test for this mutant generated by *LLMutantKiller*. The generated test invokes `toString` on the mutated function `asec` to obtain its body and asserts on the presence of the original code fragment `this['re']` to distinguish the mutant from the original code. To discourage this, we added an explicit instruction to the system prompt to discourage source code inspection. Nevertheless, LLMs cannot be trusted to follow instructions, so we cannot rule out this phenomenon. To understand the prevalence of this phenomenon, Section 3 reports a manual inspection of all mutant-killing tests produced by *LLMutantKiller* to assess behavioral validity.

3 Evaluation

This evaluation aims to answer the following Research Questions:

- **RQ1** How effective is *LLMutantKiller* at generating tests that kill mutants?
- **RQ2** How does including existing tests in prompts affect *LLMutantKiller*'s effectiveness?

Application	Description	Weekly Downloads	#LOC	#Tests	Coverage		StrykerJS						
					Stmt	Branch	Total	Killed	Survived	Timeout	Mut. Score	Time (s)	Req. Sample
Complex.js	complex numbers	2.54M	1,425	216	71.82%	67.54%	1,302	763	539	0	58.60	544	225
countries-and-timezones	accessing countries and timezones data	637.16K	165	58	100%	92.55%	140	134	6	0	95.71	96	6
crawler-url-parser	URL parser for crawling	46	209	185	96.39%	91.25%	226	141	85	0	62.39	545	70
delta	Format for representing rich text documents and changes	4.86M	806	180	98.99%	95.89%	834	686	88	60	89.45	3,452	72
image-downloader	downloading image to disk from a given URL	12.64K	64	11	100%	93.75%	43	28	11	4	74.42	395	11
node-dirty	key value store with append-only disk log	3,967	207	37	83.01%	71.15%	160	104	56	0	65.00	346	49
node-geo-point	calculations involving geographical coordinates	3,306	406	10	85.36%	70.58%	158	98	60	0	62.03	242	53
node-jsonfile	reading/writing JSON files	108.26M	102	43	97.87%	94.11%	61	54	5	2	91.80	577	5
plural	plural forms of nouns	824	103	14	95.38%	72.72%	180	143	37	0	79.44	65	34
pull-stream	pipeable pull-stream	78.29K	602	364	90.96%	80.84%	474	318	116	40	75.53	1,760	90
q	promises	13.42M	2,111	243	89.5%	70.92%	1,058	105	927	26	12.38	7,946	272
spacl-core	path-based access control	1	377	38	100%	100%	259	239	20	0	92.28	379	20
zip-a-folder	zip/tar utility	175.04K	156	22	100%	96.87%	74	38	8	28	89.19	1,332	8

Table 1. Subject applications used to evaluate *LLMutantKiller*.

- **RQ3** How does the effectiveness of *LLMutantKiller* depend on the underlying LLM?
- **RQ4** How often does *LLMutantKiller* produce tests that are invalid?
- **RQ5** How does the choice of mutation operators affect *LLMutantKiller*'s ability to kill mutants?
- **RQ6** How does *LLMutantKiller*'s ability to kill mutants compare to that of *TestPilot*, a state-of-the-art LLM-based unit test generation tool?
- **RQ7** What is the cost of running *LLMutantKiller*?

Below, we describe our experimental methodology and then address each research question.

3.1 Experimental Methodology

Selecting subject applications. Our goal is to evaluate *LLMutantKiller* on real-world JavaScript/TypeScript projects that reflect a variety of domains and programming styles. To that end, we considered all 25 projects that were used to evaluate *LLMorpheus*, an LLM-based mutation testing tool for JavaScript/TypeScript [52]. We reused this pool because it had already been curated for evaluating JavaScript/TypeScript mutation-testing techniques and spans several project domains and programming styles. Since RQ6 involves running *TestPilot* [47] on subject applications, we discarded any subject application on which *TestPilot* could not be run successfully. After discarding subject applications that do not run with *StrykerJS* (v9.6.1) or with Node.js (v20.19.4), we were left

with 13 applications. Table 1 shows, for each of these applications: a brief description, number of weekly downloads, number of lines of code, number of tests, and statement and branch coverage achieved by their existing test suites. The remaining columns in the table show results obtained by running *StrykerJS* on these applications: the total number of mutants produced, the number of mutants that were killed, the number of mutants that survived, the number of mutants that timed out, the mutation score[1] reported by *StrykerJS*, and the running time of *StrykerJS* in seconds. The final column reports the sample size for each application used in our evaluation, as described below.

Selecting surviving mutants. As can be seen in the table, the number of surviving mutants (i.e., not killed by the project's original test suite) varies considerably across applications, ranging from 5 surviving mutants for *node-jsonfile* to 927 for *q*. Answering RQ1 requires determining, for each mutant *not* killed by *LLMutantKiller*, whether it could be killed, which involves complex non-local manual analysis. Since such analysis can be very time-consuming, we evaluated a randomly selected sample rather than all surviving mutants. For each project, we treated its surviving mutants as a finite population and used Cochran's sample-size formula for proportions with finite-population correction [10] to determine the required sample size. We used a 95% confidence level, a 5% margin of error, and the conservative population-proportion estimate $p = 0.5$, which maximizes the required sample size when the true proportion is unknown. The resulting per-project sample sizes are shown in the last column of Table 1. We then randomly sampled that many surviving mutants from each project, yielding 915 surviving mutants for evaluating *LLMutantKiller*.

Classifying mutants as behavioral changes or equivalent. To understand *LLMutantKiller*'s effectiveness comprehensively, it is necessary to determine whether a mutant could be killed. When *LLMutantKiller* produces a valid mutant-killing test, that test serves as a witness that the mutant is not equivalent to the original program; mutants for which no such test is produced require separate manual classification. Toward that end, we used the following process. We began by running *LLMutantKiller* using all configurations and observing the mutants that it killed. To guard against LLMs "cheating" (e.g., inspecting source code instead of exercising behavior), we manually reviewed all mutant-killing tests and retained only those deemed valid (see below for details on the process used for classifying tests). Then, any mutant not killed by the remaining valid tests was reviewed and classified as either a behavioral change or as an equivalent mutant.

To reduce subjectivity in the process of classifying mutants, two authors independently labeled the mutants for which *LLMutantKiller* produced no valid mutant-killing test following a brief calibration session in which they jointly reviewed a small set of mutants outside the evaluation set. This process demonstrated strong agreement (Cohen's $\kappa = 0.96$ [11]). The authors then met to resolve the remaining disagreements and reached consensus for all mutants. To further support this classification, the authors manually wrote tests to kill all mutants that were classified as behavioral changes. Following this process, of the 915 surviving mutants under consideration, 815 mutants were classified as behavioral changes and 100 as equivalent mutants.

Classifying tests as valid or invalid. LLMutantKiller's *test validator* automatically rejects candidate tests based on syntactic validity, dependency constraints, and differential execution behavior (test should pass on the original program and fail on the mutant). However, these automated checks are insufficient for behavioral validity, since LLMs may produce tests that exploit non-behavioral signals (e.g., source-text inspection as was illustrated in Figure 6). Therefore, we further manually inspected and validated the mutant-killing tests generated by *LLMutantKiller*. Similar to the classification of mutants as behavioral changes or equivalent mutants, two authors independently classified

[1]The mutation score aims to provide a measure of test suite quality and is calculated as the fraction of the total number of mutants that are detected (i.e., killed or timed out), see https://stryker-mutator.io/docs/General/faq/.

each test as valid or invalid after a short session in which they jointly examined a small, separate calibration set (two mutant-killing tests per subject application) drawn from preliminary runs and disjoint from the evaluation set. The authors then met and resolved disagreements through discussion. In the experimental results presented below, we report both the number of invalid tests and the level of inter-rater agreement (Cohen's κ) between the authors who classified the tests.

LLM and prompting details. We experimented with three LLMs: Anthropic's *claude-sonnet-4.6* (date: *2/17/2026*), Mistral's *devstral-2512* (date: 12/9/2025), Meta's *llama-3.3-70b-instruct* (date: 12/6/2024), using a commercial LLM service provided by *openrouter.ai*. In each case, we use temperature 0.5 and the default settings for all other hyper-parameters. We experimented with two variations of the prompt template of Figure 2: (i) **Full prompt (with tests):** includes the mutated file, mutation description, and the existing project tests, and (ii) **Prompt without tests:** identical to the full prompt but omits existing tests. By default, *LLMutantKiller* does not include a project's test suite in the prompt. To account for network problems, we configured *LLMutantKiller* to retry up to 3 times (so 4 total attempts) on 429 and 5xx errors with wait time starting from 800ms and doubling with each retry.

Iteration limit. As discussed in Section 2, *LLMutantKiller* re-prompts the LLM when an unsatisfactory candidate test file is produced, up to a specified number of times. In our experiments, we specified a limit of 10 attempts (i.e., up to 9 re-prompting attempts after the initial prompt).

Nondeterminism and Reproducibility. Since LLMs are nondeterministic, we conduct all experiments twice and report cross-run agreement as the percentage of mutants that receive the same killed/not-killed outcome across both runs. The source code for *LLMutantKiller* and full experimental results for both runs are included in the supplemental materials associated with this paper. *LLMutantKiller* and all experimental results will be released as open-source and an artifact will be submitted for evaluation.

System details. Experiments were run on a MacBook Pro with an Apple M1 Pro CPU and 16 GB RAM, running macOS 15.6.1. The pipeline was executed using *StrykerJS* (v9.6.1) and, Node.js v20.19.4 (npm 11.5.1). Model inference was performed via the *openrouter.ai* API[2], so end-to-end runtime is primarily API/network-bound rather than determined by local compute.

3.2 RQ1: How effective is *LLMutantKiller* at generating tests that kill mutants?

Table 2 shows results for an experiment with the *claude-sonnet-4.6* LLM in which we ran *LLMutantKiller* on all 915 surviving mutants, with up to 10 attempts per mutant, using *LLMutantKiller*'s default configuration, which *does not* include a project's test suite in the prompt. The table first shows, for each subject application, the number of surviving mutants under consideration, the number of mutants killed in each iteration, and the total number and percentage of mutants killed. The last three columns of the table show the running time of *LLMutantKiller* and the number of input and output tokens used, respectively. From these results, it can be seen that 811 (88.6%) of the 915 surviving mutants were killed by the generated tests. Most of these mutants (511) were killed in the first iteration. An additional 253 mutants were killed in iterations 2–5, confirming that providing the LLM with feedback is useful. After that, diminishing returns are observed in iterations 6-10. On average, *LLMutantKiller* generated 1.94 tests for each surviving mutant that was killed. To illustrate the potential practical impact, extrapolating from the sample of surviving mutants in q, the project's mutation score is estimated to increase from 12.38% to approximately 92.9%. Full results are included in the supplemental materials.

[2]See https://openrouter.ai.

Project Name	#surviving mutants	#surviving mutants killed											cost		
		it.#1	it.#2	it.#3	it.#4	it.#5	it.#6	it.#7	it.#8	it.#9	it.#10	total (%)	time (sec)	#tokens-in	#tokens-out
Complex.js	225	131	22	10	14	11	4	2	1	0	2	197 (87.6%)	23,181	11,462,641	1,191,182
countries-and-timezones	6	4	0	0	0	0	0	1	0	0	0	5 (83.3%)	2,747	797,036	203,201
crawler-url-parser	70	30	12	2	3	2	3	2	1	1	1	57 (81.4%)	16,222	4,233,422	1,215,194
delta	72	31	12	6	3	1	3	0	1	1	2	60 (83.3%)	30,958	10,735,660	2,715,254
image-downloader	11	6	4	0	1	0	0	0	0	0	0	11 (100.0%)	567	38,798	19,685
node-dirty	49	9	11	11	6	2	2	1	0	0	0	42 (85.7%)	11,470	2,525,852	774,496
node-geo-point	53	42	3	4	3	1	0	0	0	0	0	53 (100.0%)	804	309,423	18,420
node-jsonfile	5	2	1	0	0	1	0	0	0	0	0	4 (80.0%)	697	86,462	22,724
plural	34	26	0	2	1	0	0	1	0	1	0	31 (91.2%)	2,045	427,237	72,975
pull-stream	90	55	19	3	0	0	1	0	0	1	0	79 (87.8%)	9,722	2,096,472	675,406
q	272	159	43	20	14	2	3	4	3	1	1	250 (91.9%)	40,992	19,035,672	2,825,408
spacl-core	20	10	1	1	0	0	1	1	1	0	0	15 (75.0%)	5,876	1,010,505	238,400
zip-a-folder	8	6	0	1	0	0	0	0	0	0	0	7 (87.5%)	704	98,061	26,871
Total	915	511	128	60	45	20	17	12	7	5	6	**811 (88.6%)**	145,985	52,857,241	9,999,216

Table 2. Mutants killed by *LLMutantKiller* using *claude-sonnet-4.6*; Prompt without existing tests; Inter-rater reliability of test validity labels: Cohen's $\kappa = 0.89$ [11]; Valid mutant-killing tests on non-equivalent mutants: 777/815 (**95.3%**).

Project Name	#surviving mutants	#surviving mutants killed											cost		
		it.#1	it.#2	it.#3	it.#4	it.#5	it.#6	it.#7	it.#8	it.#9	it.#10	total (%)	time (sec)	#tokens-in	#tokens-out
Complex.js	225	129	24	16	12	5	3	2	0	3	1	195 (86.7%)	21,142	12,080,584	1,139,743
countries-and-timezones	6	4	0	0	0	0	0	0	1	0	0	5 (83.3%)	1,010	396,582	52,689
crawler-url-parser	70	34	8	7	1	5	1	1	1	0	0	58 (82.9%)	9,958	8,749,540	466,130
delta	72	5	17	7	4	1	1	1	1	1	1	39 (54.2%)	24,700	15,720,878	1,440,781
image-downloader	11	4	2	4	1	0	0	0	0	0	0	11 (100.0%)	419	101,810	12,384
node-dirty	49	7	13	9	5	5	2	0	4	0	1	46 (93.9%)	10,145	2,771,351	735,158
node-geo-point	53	42	7	3	1	0	0	0	0	0	0	53 (100.0%)	874	378,618	18,082
node-jsonfile	5	2	1	1	0	0	1	0	0	0	0	5 (100.0%)	474	153,243	24,204
plural	34	24	3	0	0	1	0	2	1	0	0	31 (91.2%)	2,062	822,978	74,629
pull-stream	90	28	40	5	0	2	2	0	0	0	0	77 (85.6%)	9,219	5,123,251	621,649
q	272	157	37	25	15	5	5	8	2	2	2	258 (94.9%)	36,126	35,779,275	2,298,616
spacl-core	20	6	3	1	0	1	0	1	1	0	0	13 (65.0%)	3,648	2,013,676	126,253
zip-a-folder	8	2	3	0	0	0	0	0	1	0	0	6 (75.0%)	2,743	1,126,688	174,440
Total	915	444	158	78	39	25	15	15	12	6	5	**797 (87.1%)**	122,520	85,218,474	7,184,758

Table 3. Mutants killed by *LLMutantKiller* using *claude-sonnet-4.6*; Full prompt (with existing tests); Inter-rater reliability of test validity labels: Cohen's $\kappa = 0.93$ [11]; Valid mutant-killing tests on non-equivalent mutants: 763/815 (**93.6%**).

Using Anthropic's *claude-sonnet-4.6* LLM and its default configuration, *LLMutantKiller* generated tests that killed 811 of the 915 surviving mutants under consideration.

To assess *LLMutantKiller*'s effectiveness more comprehensively, we must account for (i) which surviving mutants are behavioral changes vs. equivalent and (ii) whether the produced mutant-killing tests are behaviorally valid. Following the process described in Subsection 3.1, we manually inspected the mutant-killing tests and identified 34 invalid cases. Hence, *LLMutantKiller* produced valid tests that killed 777 (95.3%) of the 815 non-equivalent mutants.

Focusing on the 815 non-equivalent surviving mutants and excluding the 34 cases where mutants were killed only by invalid tests, *LLMutantKiller* generated valid tests that killed 777 (95.3%) of the 815 non-equivalent surviving mutants.

Project Name	#surviving mutants	#surviving mutants killed											cost		
		it.#1	it.#2	it.#3	it.#4	it.#5	it.#6	it.#7	it.#8	it.#9	it.#10	total (%)	time (sec)	#tokens-in	#tokens-out
Complex.js	225	6	46	26	23	4	3	4	4	2	3	121 (53.8%)	10,729	17,816,898	213,249
countries-and-timezones	6	1	1	1	0	0	0	0	0	0	1	4 (66.7%)	716	182,725	7,088
crawler-url-parser	70	9	10	3	7	3	0	1	0	0	0	33 (47.1%)	4,187	1,895,707	73,161
delta	72	6	13	4	5	0	1	1	0	1	1	32 (44.4%)	7,510	3,015,668	80,674
image-downloader	11	2	1	3	0	1	0	0	0	0	0	7 (63.6%)	901	260,920	13,990
node-dirty	49	10	6	3	1	0	0	2	0	0	0	22 (44.9%)	5,625	1,687,520	120,329
node-geo-point	53	34	6	2	0	1	0	0	0	1	1	45 (84.9%)	1,459	687,367	20,872
node-jsonfile	5	0	0	1	0	0	0	0	0	0	0	1 (20.0%)	824	118,178	9,004
plural	34	6	2	8	1	3	1	0	1	0	0	22 (64.7%)	1,804	566,840	17,554
pull-stream	90	7	26	7	1	2	4	1	1	0	0	49 (54.4%)	5,911	1,929,379	110,417
q	272	2	7	109	18	10	5	2	5	1	2	161 (59.2%)	17,154	30,123,118	381,854
spacl-core	20	7	0	0	0	0	0	0	0	0	0	7 (35.0%)	1,769	420,713	16,578
zip-a-folder	8	0	2	0	1	1	1	1	0	0	0	6 (75.0%)	914	209,724	15,564
Total	915	90	120	167	57	25	15	12	11	5	8	**510 (55.7%)**	59,503	58,914,757	1,080,334

Table 4. Mutants killed by *LLMutantKiller* using *devstral-2512*; Prompt without existing tests; Inter-rater reliability of test validity labels: Cohen's $\kappa = 1.00$ [11]; Valid mutant-killing tests on non-equivalent mutants: 509/815 (**62.4%**).

3.3 RQ2: How does including existing tests in prompts affect *LLMutantKiller*'s effectiveness?

Existing tests may provide examples of project-specific setup code, API usage, and execution patterns that reach or exercise code near the mutation; RQ2 evaluates whether including this information in the prompt affects *LLMutantKiller*'s effectiveness. Table 3 shows the results of an experiment with *claude-sonnet-4.6* using a prompt that includes the subject application's test suite. The total number of mutants killed is very similar (797 with tests vs. 811 without tests). Similar to the previous experiment, most mutants are killed in the first few iterations, and manual inspection revealed 34 invalid tests. Table 3 also shows lower total running time in this configuration (122,520 seconds vs. 145,985 seconds without tests), although it uses more input tokens as discussed in RQ7.

Including existing tests in the prompt does not substantially change the number of surviving mutants killed: using *claude-sonnet-4.6* with a prompt that includes the application's tests, *LLMutantKiller* produced valid tests that kill 763 (93.6%) of the 815 non-equivalent surviving mutants.

Next, we compared the set S_1 of mutants killed by valid tests in the experiment without tests (Table 2) and the set S_2 of mutants killed in the experiment with tests (Table 3). We observed that: $S_1 \cap S_2$ contains 742 mutants that are killed in both configurations, $S_1 \setminus S_2$ contains 35 mutants killed only in configuration S_1, that $S_2 \setminus S_1$ contains 21 mutants killed only in configuration S_2, and that $S_1 \cup S_2$ contains 798 mutants.

Different mutants are killed depending on whether an application's test suite is included in the prompt, suggesting that it is worthwhile to use both prompts. Using *claude-sonnet-4.6*, such a combined approach enables *LLMutantKiller* to generate valid tests that kill 798 (97.9%) of the non-equivalent surviving mutants.

Future work discussed in Section 6 will include the exploration of techniques for improving *LLMutantKiller*'s effectiveness by combining different prompts.

Project Name	#surviving mutants	#surviving mutants killed											cost		
		it.#1	it.#2	it.#3	it.#4	it.#5	it.#6	it.#7	it.#8	it.#9	it.#10	total (%)	time (sec)	#tokens-in	#tokens-out
Complex.js	225	5	2	6	1	3	6	1	0	2	0	26 (11.6%)	39,143	28,721,525	654,654
countries-and-timezones	6	0	1	2	0	0	1	0	0	0	0	4 (66.7%)	938	95,901	4,136
crawler-url-parser	70	0	1	4	0	3	0	0	1	0	0	9 (12.9%)	15,860	4,039,535	317,092
delta	72	0	6	5	1	2	1	1	2	1	0	19 (26.4%)	12,396	4,473,012	186,002
image-downloader	11	2	2	0	0	1	0	0	0	0	0	5 (45.5%)	1,177	274,339	13,888
node-dirty	49	2	3	0	1	3	1	2	0	0	0	12 (24.5%)	7,881	2,266,493	112,608
node-geo-point	53	22	16	3	2	0	1	0	0	0	0	44 (83.0%)	3,155	791,744	44,091
node-jsonfile	5	0	0	0	0	1	1	0	0	1	0	3 (60.0%)	656	110,085	7,832
plural	34	1	11	3	2	1	0	0	0	0	0	18 (52.9%)	2,906	736,561	25,019
pull-stream	90	0	13	9	6	4	2	1	1	1	0	37 (41.1%)	15,020	2,934,873	180,189
q	272	0	7	6	7	5	5	0	2	1	1	34 (12.5%)	63,864	47,464,789	1,202,739
spacl-core	20	3	1	1	0	0	0	0	0	0	0	5 (25.0%)	2,659	559,679	20,177
zip-a-folder	8	0	0	0	1	0	0	1	0	0	0	2 (25.0%)	1,270	343,089	16,857
Total	915	35	63	39	21	23	18	6	6	6	1	**218 (23.8%)**	166,925	92,811,625	2,785,284

Table 5. Mutants killed by *LLMutantKiller* using *llama-3.3-70b-instruct*; Prompt without existing tests; Inter-rater reliability of test validity labels: Cohen's $\kappa = 0.79$ [11]; Valid mutant-killing tests on non-equivalent mutants: 216/815 (**26.5%**).

3.4 RQ3: How does the effectiveness of *LLMutantKiller* depend on the underlying LLM?

Here, we report on experiments with two additional LLMs: *devstral-2512* and *llama-3.3-70b-instruct*. In these experiments, *LLMutantKiller*'s default prompt configuration without the subject application's tests was used. Results for these experiments can be found in Tables 4 and 5. As can be seen in these tables, a total of 510 surviving mutants are killed when using the *devstral-2512* model and only 218 are killed when using the *llama-3.3-70b-instruct* model. Moreover, the results in Tables 4 and 5 suggest that *devstral-2512* and *llama-3.3-70b-instruct* converge more slowly and need more iterations to kill mutants than *claude-sonnet-4.6*.

Manual inspection of the generated tests revealed that 2 mutants were killed only by invalid tests with *devstral-2512* and that 1 mutant was killed only by invalid tests with *llama-3.3-70b-instruct*. To account for LLM nondeterminism, we conducted the experiments twice with each configuration. For *claude-sonnet-4.6*, 786 mutants were killed in both runs without tests and 838 were killed in at least one run (cross-run agreement=94.3%); with tests, 758 mutants were killed in both runs and 831 were killed in at least one run (cross-run agreement=92.0%). For *devstral-2512* (without tests), 457 mutants were killed in both runs and 572 were killed in at least one run (cross-run agreement=87.4%). For *llama-3.3-70b-instruct* (without tests), 143 mutants were killed in both runs and 298 were killed in at least one run (cross-run agreement=83.1%).

The choice of LLM substantially affects the results. Using *LLMutantKiller*'s default configuration, *claude-sonnet-4.6* performs best with 777 (95.3%) non-equivalent surviving mutants killed by valid tests, followed by *devstral-2512* with 509 (62.4%) and *llama-3.3-70b-instruct* with 216 (26.5%). This ranking is consistent across 2 repeats.

3.5 RQ4: How often does *LLMutantKiller* produce tests that are invalid?

As described in Section 3.1, we manually assessed the behavioral validity of each mutant-killing test produced by *LLMutantKiller*. Inter-rater agreement, measured using Cohen's κ [11], remained high across all configurations. We identified 34 invalid mutant-killing tests for *claude-sonnet-4.6* without tests (Cohen's $\kappa = 0.89$), 34 for *claude-sonnet-4.6* with tests (Cohen's $\kappa = 0.93$), 2 for *devstral-2512* (Cohen's $\kappa = 1.00$), and 1 for *llama-3.3-70b-instruct* (Cohen's $\kappa = 0.79$).

Mutator	Syntactic rule	Sample	Equiv.	Valid kills
Arithmetic Operator	`e1 + e2 ↔ e1 - e2`; `e1 * e2 ↔ e1 / e2`; `e1 % e2 ↦ e1 * e2`.	69 (7.5%)	1 (1.4%)	66/68 (97.1%)
Array Declaration	`[e1, ..., en] ↦ []`; `[] ↦ ["Stryker was here"]`; `new Array(e1, ..., en) ↦ new Array()`.	15 (1.6%)	3 (20.0%)	12/12 (100.0%)
Arrow Function	`args => e ↦ () => undefined`.	2 (0.2%)	1 (50.0%)	1/1 (100.0%)
Assignment Operator	`x += e ↔ x -= e`; `x *= e ↔ x /= e`; `x %= e ↦ x *= e`; `x «= e ↔ x »= e`; `x &= e ↔ x |= e`; `x &&= e ↔ x ||= e`; `x ??= e ↦ x &&= e`.	0 (0.0%)	0 (–)	–
Block Statement	`{ s1; ...; sn; } ↦ {}`.	158 (17.3%)	6 (3.8%)	144/152 (94.7%)
Boolean Literal	`true ↦ false`; `false ↦ true`; `!e ↦ e`.	42 (4.6%)	14 (33.3%)	25/28 (89.3%)
Conditional Expression	`if (e) ↦ if (true/false)`; loop test `e ↦ false`; `e_bool ↦ true/false`; `for (;;) ↦ for (; false ;)`; `case v: s* ↦ case v:`.	290 (31.7%)	29 (10.0%)	244/261 (93.5%)
Equality Operator	`e1 == e2 ↔ e1 != e2`; `e1 === e2 ↔ e1 !== e2`; `e1 < e2 ↦ {e1 <= e2, e1 >= e2}`; similarly for `<=`, `>`, and `>=`.	82 (9.0%)	13 (15.9%)	67/69 (97.1%)
Logical Operator	`e1 && e2 ↦ e1 || e2`; `e1 || e2 ↦ e1 && e2`; `e1 ?? e2 ↦ e1 && e2`.	43 (4.7%)	3 (7.0%)	39/40 (97.5%)
Method Expression	`e.trim() ↦ e`; `Math.min(args) ↦ Math.max(args)`; similarly for selected string, array, Math, and Date methods.	2 (0.2%)	0 (0.0%)	2/2 (100.0%)
Object Literal	`{p1: e1, ..., pn: en} ↦ {}`.	11 (1.2%)	1 (9.1%)	10/10 (100.0%)
Optional Chaining	`e?.p ↦ e.p`; `e?.[i] ↦ e[i]`; `e?.(args) ↦ e(args)`.	0 (0.0%)	0 (–)	–
Regex	`/r/ ↦ /r'/`; `new RegExp("r") ↦ new RegExp("r'")`, where `r'` is a level-1 mutation of the regular-expression AST.	43 (4.7%)	10 (23.3%)	32/33 (97.0%)
String Literal	`"s" ↦ ""`; `"" ↦ "Stryker was here!"`; similarly for template literals.	139 (15.2%)	12 (8.6%)	124/127 (97.6%)
Unary Operator	`-e ↦ +e`; `+e ↦ -e`; `~e ↦ e`.	15 (1.6%)	6 (40.0%)	9/9 (100.0%)
Update Operator	`x++ ↦ x--`; `x-- ↦ x++`; `++x ↦ --x`; `--x ↦ ++x`.	4 (0.4%)	1 (25.0%)	2/3 (66.7%)
Total	–	**915**	**100**	**777/815 (95.3%)**

Table 6. Mutators supported by *StrykerJS* v9.6.1.

Manual validation indicates that *LLMutantKiller* rarely produces behaviorally invalid mutant-killing tests: at most 3.7% of the mutants are killed only by invalid tests in any configuration.

3.6 RQ5: How does the choice of mutation operators affect *LLMutantKiller*'s ability to kill mutants?

To answer RQ5, we analyze *LLMutantKiller*'s results across the mutators (i.e., classes of mutation operators; see Table 6 for examples) supported by *StrykerJS*, a state-of-the-art rule-based mutation testing tool for JavaScript and TypeScript [50]. Table 6 reports, for each mutator, the syntactic rewrite rule, frequency in our evaluation sample, number of sampled mutants classified as equivalent, and valid-kill rate achieved by *LLMutantKiller*'s default configuration (*claude-sonnet-4.6* without existing tests in the prompt).

The per-mutator valid-kill rates are at least 89.3% for every mutator with at least 10 sampled mutants, suggesting that the remaining failures are not concentrated in one mutator. To check whether these differences reflect a systematic relationship, we compare valid-kill outcomes across mutators using a permutation test and report Cramer's V to quantify the strength of the relationship [14, 19]. For the default configuration, the association is weak and not statistically significant ($V = 0.134$, $p = 0.301$). Across the other configurations, the association remains weak in magnitude; full per-configuration results are included in the supplemental materials.

The mutator used to create a mutant has a statistically significant but weak association with whether that mutant is classified as equivalent or non-equivalent ($V = 0.267$, $p < 0.001$). Among mutators with at least 10 sampled mutants, *Unary Operator*, *Boolean Literal*, and *Regex* have higher equivalent-mutant rates than *Arithmetic Operator* and *Block Statement*. This weak association is expected because equivalence depends not only on the syntactic rewrite rule, but also on the surrounding program semantics, reachability, and project context. Consequently, mutator alone does not explain which surviving mutants are equivalent or remain unkilled by *LLMutantKiller*.

Project Name	#mutants	#mutants killed
Complex.js	225	70
countries-and-timezones	6	0
crawler-url-parser	70	1
delta	72	0
image-downloader	11	0
node-dirty	49	1
node-geo-point	53	0
node-jsonfile	5	0
plural	34	0
pull-stream	90	1
q	272	4
spacl-core	20	0
zip-a-folder	8	0
Total:	915	77

Table 7. Number of mutants killed by tests generated by TestPilot.

Using Anthropic's *claude-sonnet-4.6* LLM and its default configuration, mutator has only a weak, statistically insignificant association with valid-kill outcome. Thus, no single mutator explains the remaining failures. The mutator used to create a mutant is weakly associated with whether that mutant is classified as equivalent or non-equivalent, despite statistical significance.

3.7 RQ6: How does *LLMutantKiller* compare with a state-of-the-art LLM-based test generation tool?

To answer RQ6, we compare *LLMutantKiller* against *TestPilot* [47] on the same set of surviving mutants. For each project, we perform the following steps: (i) run TestPilot[3] to generate a test suite, (ii) execute the generated tests on the original and mutated versions, and (iii) determine the number of mutants killed using the tests. Table 7 shows the results. As can be seen in the table, only 77 of the 915 surviving mutants are killed by tests created by *TestPilot*, compared to 777 killed using valid tests produced by *LLMutantKiller* in its default configuration using *claude-sonnet-4.6*.

Using tests generated by *TestPilot*, a general-purpose LLM-based unit test generation tool, only 77 of the 915 surviving mutants are killed compared to 777 killed using valid tests produced by *LLMutantKiller* in its default configuration using *claude-sonnet-4.6*. This confirms the usefulness of techniques for test generation that specifically target the problem of killing mutants.

3.8 RQ7: What is the cost of running *LLMutantKiller*?

To assess the cost of running *LLMutantKiller*, we consider the running time of the tool and the number of tokens used. The latter serves as a proxy for the cost of LLM usage given that LLM charges are generally calculated as a function of the number of input tokens and output tokens.

Running time. Based on the running times shown in Table 2, the average per-mutant running time of *LLMutantKiller* in its default configuration without tests included in the prompt ranges from 15.2 seconds for *node-geo-point* to 457.8 seconds for *countries-and-timezones*. Running *LLMutantKiller* on all projects sequentially requires 145,985 seconds. This time is dominated by communication with the LLM. Including tests in the prompts reduces total running time to 122,520 seconds (see Table 3). The total running time for *LLMutantKiller* with *devstral-2512* and *llama-3.3-70b-instruct* is 59,503 and 166,925 seconds, respectively (Tables 4 and 5).

[3]In our experiments, we used *TestPilot* v2[4], using its default configuration.

Token usage. As can be seen in Table 2, using *claude-sonnet-4.6* and using its default configuration, *LLMutantKiller* requires a total of 52,857,241 input tokens and 9,999,216 output tokens. At the time of writing, the cost of *claude-sonnet-4.6* at *openrouter.ai* is $3 per million input tokens and $15 per million output tokens, so the entire experiment would cost approximately $308.55. However, Claude models also support prompt caching for input tokens; at the time of writing, *openrouter.ai* charges $3.75 per million cache-write tokens and $0.30 per million cache-read tokens for *claude-sonnet-4.6*, while output tokens use the standard rate. With prompt caching enabled, this experiment cost $253.14, saving $55.42. Full token accounting, including cache-write and cache-read tokens, is included in the supplemental materials. The experiment with prompts that include the subject application's tests (see Table 3) requires 85,218,474 input tokens and 7,184,758 output tokens. Here, the number of input tokens is about 1.6x higher because including the tests in the prompts involves significant token usage, bringing the total estimated cost to approximately $363.42. With prompt caching enabled, this experiment cost $244.92, saving $118.51.

When using *devstral-2512* (Table 4), a total of 58,914,757 input tokens and 1,080,334 output tokens are used. At the time of writing, the cost of *devstral-2512* at *openrouter.ai* is $0.4 per million input tokens and $2 per million output tokens, so the entire experiment cost approximately $25.72.

When using *llama-3.3-70b-instruct* (Table 5), a total of 92,811,625 input tokens and 2,785,284 output tokens are used. At the time of writing, the cost of *llama-3.3-70b-instruct* at *openrouter.ai* is $0.10 per million input tokens and $0.32 per million output tokens, so the entire experiment cost approximately $10.17.

Note that token usage tends to increase when fewer mutants are killed because *LLMutantKiller* will stop re-prompting when a mutant is killed. In other words, for mutants that are not killed, the full 10 iterations are used. Since *LLMutantKiller* kills fewer mutants when *devstral-2512* and especially *llama-3.3-70b-instruct* are used, more iterations are needed, incurring higher token usage.

LLMutantKiller's average running time ranges from 65.0 (*devstral-2512*) to 182.4 (*llama-3.3-70b-instruct*) seconds per mutant, depending on the configuration and LLM being used. Including existing tests substantially increases input-token usage (1.6x higher when using *claude-sonnet-4.6*). When excluding existing tests, input-token usage varies across LLMs, ranging from 52,857,241 (*claude-sonnet-4.6*) to 92,811,625 (*llama-3.3-70b-instruct*). Output-token usage ranges from 1,080,334 (*devstral-2512*) to 9,999,216 (*claude-sonnet-4.6*). Across configurations, the estimated total cost ranges from $10.17 (*llama-3.3-70b-instruct*, without tests) to $253.14 (*claude-sonnet-4.6*, without tests).

4 Threats to Validity

The results presented in Section 3 may be impacted by selection bias in several ways. First, the set of subject projects may not be fully representative of all JavaScript and TypeScript applications. However, these projects have been used in prior work on mutation testing and automated test generation [4, 47, 52] and span multiple application domains and programming styles.

Second, for each project, we randomly samples surviving mutants to assess *LLMutantKiller*'s ability to kill surviving mutants. This was necessary because the large number of surviving mutants produced by *StrykerJS* (1,956 in total), which makes exhaustive evaluation prohibitively expensive given the manual validation steps needed to answer RQ1—RQ5. To reduce sampling bias, we used Cochran's sample-size formula for proportions with finite-population correction [10] (95% confidence, 5% margin of error, and conservative population-proportion estimate $p = 0.5$) to determine the required per-project sample sizes. The resulting samples support estimating proportions over the surviving mutants in each studied project, but may still miss specific mutant characteristics.

Third, we evaluate *LLMutantKiller*'s effectiveness on surviving mutants produced by *StrykerJS* [50] and our findings may be influenced by the specific mutation operators and design choices of this tool. *StrykerJS* is a state-of-the-art tool that implements standard mutation operators. To assess the extent to which our results depend on these operators, we analyzed the relationship between mutator and *LLMutantKiller*'s valid-kill outcome, finding only a weak association. However, our results may not generalize to mutants produced by other tools or operator sets.

Fourth, the effectiveness of *LLMutantKiller* depends on the ability of the underlying LLM to reason about program behavior. To mitigate the potential for bias, we reported on experiments with three different LLMs (*claude-sonnet-4.6*, *devstral-2512*, and *llama-3.3-70b-instruct*). Of these, the latter two are open-weight models, which facilitates reproducibility. In our evaluation, the number of test-generation attempts per mutant is limited to 10 attempts. This choice reflects a practical trade-off between effectiveness and cost. However, it is possible that *LLMutantKiller* could kill more surviving mutants if given additional attempts, especially in the case of *devstral-2512* and *llama-3.3-70b-instruct*, which kill far fewer mutants than *claude-sonnet-4.6* with the given budget.

LLMs exhibit inherent nondeterminism, which may affect test generation outcomes. To mitigate this threat, we conducted all experiments twice for each configuration. We also report cross-run agreement, which was at least 83.1% across all configurations. The results for both runs appear in the supplemental materials. Nondeterministic project behavior or flaky generated tests could also affect whether a mutant is judged killed or equivalent. We did not observe evidence of flaky generated tests during manual validation, but this remains a possible threat in larger deployments.

The validation of the results presented in Section 3 relies on human judgment to classify generated tests as valid or invalid, and mutants not killed by *LLMutantKiller* as killable or equivalent. In each case, human error may have impacted the classification. To mitigate subjectivity, two authors labeled items independently, and the few remaining disagreements were resolved through discussion until consensus was reached. We report inter-rater reliability using Cohen's κ.

We report cost using token usage and wall-clock execution time. Token counts provide an abstract measure of LLM usage, whereas runtime may vary due to external factors such as system load and server delays. To account for this, we conducted the experiments twice per configuration.

Lastly, our work targets JavaScript and TypeScript applications that run with Node.js and produces tests that run using the Jest testing framework. While our approach is conceptually applicable beyond this ecosystem, results may not generalize directly to other languages, testing frameworks, or build systems without additional engineering effort.

5 Related Work

This section reviews prior work on (i) traditional (non-LLM-based) techniques for killing mutants and detecting equivalent mutants, (ii) LLM-based test generation, (iii) LLM-based mutation testing, and (iv) LLM-based techniques for detecting equivalent mutants. For general background on mutation testing, the reader is referred to surveys by Jia and Harman [24] and Papadakis et al. [43].

5.1 Traditional Techniques.

The research community has explored various traditional (i.e., non-LLM-based) techniques for generating tests that kill surviving mutants. This includes techniques based on fuzzing [32], symbolic execution [8], assertion amplification [18], and multi-objective search algorithms [54]. Du et al. [17] present an empirical study that investigates how mutants are killed.

Determining whether a mutant is equivalent is an undecidable problem. Schuler and Zeller estimate that 45% of mutants that are not killed are equivalent and that determining whether a single mutant is equivalent takes about 15 minutes [48]. Since equivalent mutants significantly detract from the value of mutation testing, the research community has explored a range of

techniques for detecting equivalent mutants. This includes work based on: compiler-optimization techniques [29, 37], constraint-based techniques [36, 38], and program slicing [23]. Schuler and Zeller [48] and Papadakis et al. [42] report on heuristic techniques for classifying mutants. Madeyski et al. [35] present a systematic literature review of techniques for detecting equivalent mutants.

We view LLM-based techniques for killing surviving mutants as complementary to traditional techniques for detecting equivalent mutants. One could use an LLM-based technique such as *LLMutantKiller* to filter out mutants that represent behavioral differences, allowing computationally expensive techniques such as the ones discussed above to focus on a smaller number of mutants.

5.2 LLM-based Test Generation.

Bareiß et al. [5] present a study involving a few-shot prompting approach in which an LLM is queried with the definition of a function under test, along with an example of another function (taken from the same code base) and its associated test. They report achieving slightly better coverage than Randoop [39, 40] in a limited study of 18 Java methods.

Schäfer et al. [47] present an approach where an LLM is queried with prompts that provide various details of a function under test (signature, example snippets, documentation, source code) and implemented it in a tool called *TestPilot*. In an evaluation on a set of JavaScript/TypeScript applications, Schäfer et al. report that *TestPilot* achieves 70.2% statement coverage and 52.8% branch coverage, on average. We measured *TestPilot*'s ability to produce tests that kill surviving mutants and found that it killed only 77 of 915 surviving mutants, compared to up to 777 for *LLMutantKiller*.

LeMieux et al. [33] present *CodaMosa*, an LLM-based approach for improving the effectiveness of Search-Based Software Testing (SBST) by asking an LLM to provide example test cases when coverage improvements achieved by SBST stagnate. They implement their technique on top of Mosa [41] in Pynguin [34], an SBST tool for Python. In an evaluation on 486 benchmark modules taken from 27 projects, *CodaMosa* often achieves significantly higher coverage than two baseline techniques (Mosa and CodexOnly) while reducing coverage in a few cases.

Altmayer Pizzorno and Berger [45] present *CoverUp*, an LLM-based test generation tool for Python that aims to improve test coverage. It repeatedly prompts an LLM with coverage information to focus test generation on uncovered code. *CoverUp* provides the LLM with a `get_info` tool function that it can use to solicit additional information about types or variables. Experimental results show that *CoverUp* outperforms both CodaMosa and MuTAP [15] (discussed below) in coverage.

Ryan et al. [46] present *SymPrompt*, an LLM-based test generation technique inspired by dynamic symbolic execution [21] by querying an LLM with prompts including approximate *path constraints* corresponding to execution paths. In an evaluation on open-source Python projects, the incorporation of path constraints in prompts is found to improve test coverage significantly.

Chen et al. [9] present ChatUniTest, an LLM-based test generation tool that analyzes dependency relationships to determine relevant information (such as signatures of dependent methods and classes) that is referenced by a method under test and that should be included in prompts.

Alshahwan et al. [3] present TestGen-LLM, an LLM-based tool developed at Meta for improving existing human-written tests. They report on an industrial scale deployment where it improved 11.5% of all classes to which it was applied, with 73% of its recommendations being accepted for production deployment by Meta software engineers.

The test generation techniques discussed above do not consider the presence of mutants and do not incorporate analyses of behavioral differences between the original program and mutants derived from it. While the generated tests may kill mutants by chance, our results demonstrate that techniques specifically designed to exploit these behavioral differences are substantially more effective at generating tests that kill mutants.

5.3 LLM-based Techniques for Mutation Testing.

Bareiß et al. [5] present a few-shot prompting approach to mutation testing and compare it to Major [25], a state-of-the-art rule-based tool for Java. They report that their technique produces a similar number of mutants as Major, that their tool generates the same mutants as Major in 18.5% of all cases, but that only 62.5% of these mutants are compilable.

Dakhel et al. [15] present a test generation technique for Python that aims to produce tests that kill mutants. Their approach assumes that a set of mutants is available and begins by prompting an LLM to produce an initial set of tests. The tests are run to determine what mutants they kill, and the LLM is then repeatedly queried with a prompt that includes the test, one of the surviving mutants, and a request to create a new test that kills that mutant. This process is repeated once for each surviving mutant. Two major methodological differences exist between our work and Dakhel's. First, rather than modifying existing tests to kill mutants, *LLMutantKiller* instructs the LLM to create new tests. Second, whereas *MuTAP* prompts the LLM once per surviving mutant, *LLMutantKiller* queries it up to 10 times with feedback if a generated test fails to kill the mutant. Our results show many surviving mutants are killed in iterations 2–10, confirming the value of feedback. Other differences include that *LLMutantKiller* starts with surviving mutants from *StrykerJS* excluding already-killed mutants, and the target languages differ (JavaScript/TypeScript vs. Python).

Wang et al. [53] present MutGen, an LLM-based test-generation technique that aims to improve fault-detection capability through mutation feedback. In their approach, an LLM is queried with a prompt that includes mutation feedback describing the location, mutation operator, and status (killed, survived, or uncovered) of mutants, along with instructions to produce tests that are more diverse relative to existing ones. The generated tests are then executed, and failing tests are repaired by prompting the LLM with a structured prompt containing strategies for addressing different failure types. This process is repeated iteratively until the mutation score stabilizes or a predefined iteration limit is reached. Unlike Wang et al., whose objective is to maximize the mutation score of a generated test suite, our objective is to generate tests that kill a specified set of surviving mutants. However, both approaches employ an iterative process in which an LLM is prompted with previously generated tests to obtain tests that have the desired execution behavior. The approaches also differ in the programming language under consideration (Java versus JavaScript/TypeScript) and in the mutation testing tool used to generate mutants (PITest [2] vs. StrykerJS [50]).

Bouafif et al. [7] present PRIMG, a mutation-driven test generator for Solidity smart contracts. In their approach, an LLM first creates a test using a prompt including the program under test, the targeted mutant, and existing tests. If the test is syntactically invalid or fails on the program under test, the LLM is reprompted. PRIMG considers mutant subsumption [30] and test completeness advancement probability (TCAP) [28] to prioritize mutants that are more likely to improve the overall efficiency of test generation. Compared to PRIMG, *LLMutantKiller* uses a more elaborate feedback mechanism and targets JavaScript/TypeScript rather than Solidity.

Harman et al. [22] describe Meta's Automated Compliance Hardening (ACH) system for mutation-guided LLM-based test generation, which consists of three LLM-based agents that operate in concert: (i) a fault generation component asks an LLM to introduce bugs using prior fault information, (ii) a modified version of TestGen-LLM [3] extends existing tests to detect faults, and (iii) an equivalence detector component asks an LLM to determine whether a mutant is equivalent. *LLMutantKiller* differs by targeting surviving mutants from *StrykerJS*, by using scenario-specific feedback, and by focusing on different programming languages (JavaScript/TypeScript vs. Kotlin).

LLMorpheus [52] implements an LLM-based mutation testing technique by querying an LLM with: general mutation testing background, a source fragment with a "PLACEHOLDER", the code to substitute, and a request to introduce a behavioral difference. The main difference is that *LLMorpheus*

generates mutants, whereas *LLMutantKiller* generates tests to kill surviving mutants. We currently focus on *StrykerJS* mutants; evaluating *LLMutantKiller* on LLM-generated mutants is future work.

Straubinger et al. [49] present a mutation-testing approach based on Scientific Debugging [27]. In their approach, an LLM is given background on Scientific Debugging and asked to generate natural-language hypotheses explaining a mutant's behavior. An experiment is then conducted by creating and executing a test case against the original code and the mutant, and the LLM is asked to explain the observed behavior. If the test does not exhibit the required behavior, the LLM is re-prompted. As in our work, a limit of 10 iterations is imposed. Straubinger et al. compare their technique to a baseline where the LLM is directly prompted to kill the mutant, and to Pynguin [34], finding that it significantly outperforms both. While *LLMutantKiller* employs a similar approach, key differences exist: *LLMutantKiller* employs a more fine-grained feedback mechanism using different prompts for specific failure scenarios. For example, if a generated test fails on the original code, feedback is provided immediately without running the test against the mutant. In addition, *LLMutantKiller* targets JavaScript/TypeScript rather than Python and focuses on killing surviving mutants produced by existing mutation testing tools.

Overall, prior work on LLM-based mutation testing has explored mutant generation, mutation-guided test-suite improvement, and mutation-targeted test generation. *LLMutantKiller*'s main distinguishing feature is its scenario-specific iterative feedback mechanism that re-prompts an LLM using prompts tailored to why a previously generated test was inadequate (e.g., syntax errors, failures on the original program, or failure to kill the mutant). Our results show that this targeted feedback substantially improves effectiveness, highlighting the importance of adaptive interaction strategies for LLM-based mutation testing. In addition, *LLMutantKiller* focuses on generating new tests to kill surviving mutants produced by existing JavaScript/TypeScript mutation testing tools.

5.4 LLM-based Techniques for Detecting Equivalent Mutants

Tian et al. [51] report on a comprehensive empirical study of 3,302 method-level Java mutant pairs in which they compare the effectiveness (precision, recall, F1-score) and efficiency of 10 LLMs (using several prompting strategies) for equivalent mutant detection to that of 10 prior equivalent-mutant detection techniques, including compiler-based techniques, an ML-based technique, and a tree neural network technique. Tian et al. find that their LLM-based techniques significantly outperform the previous techniques, and that the fine-tuned code embedding strategy is the most effective.

Specialized LLM-based mutant-killing test generation techniques may complement LLM-based techniques for equivalent mutant detection by filtering out mutants guaranteed not to be equivalent.

6 Conclusions and Future Work

We have presented a feedback-directed LLM-based technique for generating tests that kill surviving mutants and implemented it in *LLMutantKiller* a tool for JavaScript/TypeScript applications. In an empirical evaluation, *LLMutantKiller* was applied to 915 randomly selected surviving mutants produced by *StrykerJS*, a state-of-the-art mutation testing tool, in 13 open-source JavaScript/TypeScript applications. The results show that *LLMutantKiller* kills up to 95.3% of surviving mutants that were classified as inducing behavioral changes and that it rarely produces invalid tests.

We intend to explore the following future work. First, while *LLMutantKiller* produces relatively few invalid tests, detecting such tests is challenging given an LLM's ability to produce arbitrary code. We plan to mitigate this with automatic filters (e.g., pattern-based detectors) and/or using an LLM to assess test validity. Second, we plan to conduct a qualitative study with developers to assess the practical usefulness of the generated tests and *LLMutantKiller*'s potential for finding bugs. Third, we intend to evaluate *LLMutantKiller* on surviving mutants produced by LLM-based mutation testing techniques [52]. Finally, since our results suggest complementary strengths across different

configurations of *LLMutantKiller*, we plan to study cost-aware strategies to orchestrate multiple prompts and models, including additional prompt variants inspired by prior mutation-guided test-generation approaches such as MuTAP [15].

7 Data Availability

The code for *LLMutantKiller* and all experimental data used in this study are included in the supplementary materials associated with this paper submission. Upon acceptance, all code and data will be open-sourced and an artifact will be submitted for evaluation.

Acknowledgments

This work was supported in part by the National Science Foundation under Grant CCF-2307742. Frank Tip is a Professor of Computer Science at Northeastern University and an Amazon Scholar. The paper describes work performed at Northeastern University and is not associated with Amazon. The authors are grateful to Michelle Thalakottur for her assistance with classification of mutants and tests.

References

[1] 2026. Jest: Delightful JavaScript Testing. https://jestjs.io/. Accessed: 2026-01-29.

[2] 2026. PITest. https://pitest.org/. Accessed 5/19/2026.

[3] Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. 2024. Automated Unit Test Improvement using Large Language Models at Meta. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024, Porto de Galinhas, Brazil, July 15-19, 2024*, Marcelo d'Amorim (Ed.). ACM, 185–196. doi:10.1145/3663529.3663839

[4] Ellen Arteca, Sebastian Harner, Michael Pradel, and Frank Tip. 2022. Nessie: Automatically testing JavaScript APIs with asynchronous callbacks. In *Proceedings of the 44th International Conference on Software Engineering*. 1494–1505.

[5] Patrick Bareiß, Beatriz Souza, Marcelo d'Amorim, and Michael Pradel. 2022. Code Generation Tools (Almost) for Free? A Study of Few-Shot, Pre-Trained Language Models on Code. *CoRR* abs/2206.01335 (2022). arXiv:2206.01335 doi:10.48550/ARXIV.2206.01335

[6] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra, and Erik Meijer. 2021. What it would take to use mutation testing in industry: a study at Facebook. In *Proceedings of the 43rd International Conference on Software Engineering: Software Engineering in Practice* (Virtual Event, Spain) *(ICSE-SEIP '21)*. IEEE Press, 268–277.

[7] Mohamed Salah Bouafif, Mohammad Hamdaqa, and Edward Zulkoski. 2025. PRIMG : Efficient LLM-driven Test Generation Using Mutant Prioritization. In *Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering, EASE 2025, Istanbul, Turkey, June 17-20, 2025*, Muhammad Ali Babar, Ayse Tosun, Stefan Wagner, and Viktoria Stray (Eds.). ACM, 1107–1116. doi:10.1145/3756681.3756991

[8] Thierry Titcheu Chekam, Mike Papadakis, Maxime Cordy, and Yves Le Traon. 2021. Killing Stubborn Mutants with Symbolic Execution. *ACM Trans. Softw. Eng. Methodol.* 30, 2 (2021), 19:1–19:23. doi:10.1145/3425497

[9] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. 2024. ChatUniTest: A Framework for LLM-Based Test Generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024, Porto de Galinhas, Brazil, July 15-19, 2024*, Marcelo d'Amorim (Ed.). ACM, 572–576. doi:10.1145/3663529.3663801

[10] William G. Cochran. 1977. *Sampling Techniques* (3 ed.). John Wiley & Sons, New York.

[11] Jacob Cohen. 1960. A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement* 20, 1 (1960), 37–46. doi:10.1177/001316446002000104

[12] Henry Coles. 2024. ArcMutate: Advanced mutation testing for Java and Kotlin. https://www.arcmutate.com.

[13] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. PIT: a practical mutation testing tool for Java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 449–452. doi:10.1145/2931037.2948707

[14] Harald Cramér. 1946. *Mathematical Methods of Statistics*. Princeton University Press, Princeton, NJ.

[15] Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C. Desmarais. 2024. Effective test generation using pre-trained Large Language Models and mutation testing. *Inf. Softw. Technol.* 171 (2024), 107468.

doi:10.1016/J.INFSOF.2024.107468

[16] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* 11, 4 (1978), 34–41. doi:10.1109/C-M.1978.218136

[17] Hang Du, Vijay Krishna Palepu, and James A. Jones. 2023. To Kill a Mutant: An Empirical Study of Mutation Testing Kills. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, René Just and Gordon Fraser (Eds.). ACM, 715–726. doi:10.1145/3597926.3598090

[18] Hang Du, Vijay Krishna Palepu, and James A. Jones. 2025. Leveraging Propagated Infection to Crossfire Mutants. In *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*. IEEE, 3136–3148. doi:10.1109/ICSE55347.2025.00150

[19] Ronald A. Fisher. 1935. *The Design of Experiments*. Oliver and Boyd, Edinburgh.

[20] Gregory Gay and Alireza Salahirad. 2023. How Closely are Common Mutation Operators Coupled to Real Faults?. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 129–140. doi:10.1109/ICST57152.2023.00021

[21] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, Vivek Sarkar and Mary W. Hall (Eds.). ACM, 213–223. doi:10.1145/1065010.1065036

[22] Mark Harman, Jillian Ritchey, Inna Harper, Shubho Sengupta, Ke Mao, Abhishek Gulati, Christopher Foster, and Hervé Robert. 2025. Mutation-Guided LLM-based Test Generation at Meta. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering, FSE Companion 2025, Clarion Hotel Trondheim, Trondheim, Norway, June 23-28, 2025*, Leonardo Montecchi, Jingyue Li, Denys Poshyvanyk, and Dongmei Zhang (Eds.). ACM, 180–191. doi:10.1145/3696630.3728544

[23] Robert M. Hierons, Mark Harman, and Sebastian Danicic. 1999. Using Program Slicing to Assist in the Detection of Equivalent Mutants. *Softw. Test. Verification Reliab.* 9, 4 (1999), 233–262.

[24] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Trans. Software Eng.* 37, 5 (2011), 649–678. doi:10.1109/TSE.2010.62

[25] René Just. 2014. The Major mutation framework: efficient and scalable mutation analysis for Java. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 433–436. doi:10.1145/2610384.2628053

[26] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)* (Hong Kong, China). 654–665.

[27] Sungmin Kang, Bei Chen, Shin Yoo, and Jian-Guang Lou. 2025. Explainable automated debugging via Large Language Model-driven Scientific Debugging. *Empir. Softw. Eng.* 30, 2 (2025), 45. doi:10.1007/S10664-024-10594-X

[28] Samuel J. Kaufman, Ryan Featherman, Justin Alvin, Bob Kurtz, Paul Ammann, and René Just. 2022. Prioritizing Mutants to Guide Mutation Testing. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 1743–1754. doi:10.1145/3510003.3510187

[29] Marinos Kintis, Mike Papadakis, Yue Jia, Nicos Malevris, Yves Le Traon, and Mark Harman. 2018. Detecting Trivial Mutant Equivalences via Compiler Optimisations. *IEEE Trans. Software Eng.* 44, 4 (2018), 308–333. doi:10.1109/TSE.2017.2684805

[30] Bob Kurtz, Paul Ammann, Márcio Eduardo Delamaro, Jeff Offutt, and Lin Deng. 2014. Mutant Subsumption Graphs. In *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014 Workshops Proceedings, March 31 - April 4, 2014, Cleveland, Ohio, USA*. IEEE Computer Society, 176–185. doi:10.1109/ICSTW.2014.20

[31] Thomas Laurent, Stephen Gaffney, and Anthony Ventresque. 2022. Re-visiting the coupling between mutants and real faults with Defects4J 2.0. In *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 189–198.

[32] Jaekwon Lee, Enrico Viganò, Oscar Cornejo, Fabrizio Pastore, and Lionel C. Briand. 2023. Fuzzing for CPS Mutation Testing. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 1377–1389. doi:10.1109/ASE56229.2023.00079

[33] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 919–931. doi:10.1109/ICSE48619.2023.00085

[34] Stephan Lukasczyk and Gordon Fraser. 2022. Pynguin: Automated Unit Test Generation for Python. In *44th IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2022, Pittsburgh, PA, USA, May 22-24, 2022*. ACM/IEEE, 168–172. doi:10.1145/3510454.3516829

[35] Lech Madeyski, Wojciech Orzeszyna, Richard Torkar, and Mariusz Jozala. 2014. Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation. *IEEE Trans. Software Eng.* 40, 1 (2014), 23–42. doi:10.1109/TSE.2013.44

[36] Simona Nica and Franz Wotawa. 2012. Using Constraints for Equivalent Mutant Detection. In *Proceedings 2nd Workshop on Formal Methods in the Development of Software, WS-FMDS 2012, Paris, France, August 28, 2012 (EPTCS, Vol. 86)*, César Andrés and Luis Llana (Eds.). 1–8. doi:10.4204/EPTCS.86.1

[37] A. Jefferson Offutt and W. M. Craft. 1994. Using Compiler Optimization Techniques to Detect Equivalent Mutants. *Softw. Test. Verification Reliab.* 4, 3 (1994), 131–154. doi:10.1002/STVR.4370040303

[38] A. Jefferson Offutt and Jie Pan. 1997. Automatically Detecting Equivalent Mutants and Infeasible Paths. *Softw. Test. Verification Reliab.* 7, 3 (1997), 165–192.

[39] Carlos Pacheco and Michael D. Ernst. 2007. Randoop: Feedback-directed Random Testing for Java. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion* (Montreal, Quebec, Canada) *(OOPSLA '07)*. ACM, New York, NY, USA, 815–816. doi:10.1145/1297846.1297902

[40] Carlos Pacheco, Shuvendu K. Lahiri, and Thomas Ball. 2008. Finding Errors in .Net with Feedback-directed Random Testing. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA '08)*. ACM, New York, NY, USA, 87–96. doi:10.1145/1390630.1390643

[41] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2018. Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets. *IEEE Trans. Software Eng.* 44, 2 (2018), 122–158. doi:10.1109/TSE.2017.2663435

[42] Mike Papadakis, Márcio Eduardo Delamaro, and Yves Le Traon. 2014. Mitigating the effects of equivalent mutants with mutant classification strategies. *Sci. Comput. Program.* 95 (2014), 298–319. doi:10.1016/J.SCICO.2014.05.012

[43] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Mutation Testing Advances: An Analysis and Survey. *Adv. Comput.* 112 (2019), 275–378.

[44] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. 2022. Practical Mutation Testing at Scale: A view from Google. *IEEE Transactions on Software Engineering* 48, 10 (2022), 3900–3912. doi:10.1109/TSE.2021.3107634

[45] Juan Altmayer Pizzorno and Emery D. Berger. 2025. CoverUp: Effective High Coverage Test Generation for Python. *Proc. ACM Softw. Eng.* 2, FSE (2025), 2897–2919. doi:10.1145/3729398

[46] Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. 2024. Code-Aware Prompting: A Study of Coverage-Guided Test Generation in Regression Setting using LLM. *Proc. ACM Softw. Eng.* 1, FSE (2024), 951–971. doi:10.1145/3643769

[47] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Trans. Software Eng.* 50, 1 (2024), 85–105. doi:10.1109/TSE.2023.3334955

[48] David Schuler and Andreas Zeller. 2013. Covering and Uncovering Equivalent Mutants. *Softw. Test. Verification Reliab.* 23, 5 (2013), 353–374. doi:10.1002/STVR.1473

[49] Philipp Straubinger, Marvin Kreis, Stephan Lukasczyk, and Gordon Fraser. 2025. Mutation Testing via Iterative Large Language Model-Driven Scientific Debugging. In *IEEE International Conference on Software Testing, Verification and Validation, ICST 2025 - Workshops, Naples, Italy, March 31 - April 4, 2025*. IEEE, 358–367. doi:10.1109/ICSTW64639.2025.10962485

[50] Stryker Team. 2026. Stryker Mutator. https://stryker-mutator.io. Accessed 1/14/2026.

[51] Zhao Tian, Honglin Shu, Dong Wang, Xuejie Cao, Yasutaka Kamei, and Junjie Chen. 2024. 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*. 1733–1745.

[52] Frank Tip, Jonathan Bell, and Max Schäfer. 2025. LLMorpheus: Mutation Testing Using Large Language Models. *IEEE Trans. Software Eng.* 51, 6 (2025), 1645–1665. doi:10.1109/TSE.2025.3562025

[53] Guancheng Wang, Qinghua Xu, Lionel C. Briand, and Kui Liu. 2026. Mutation-Guided Unit Test Generation with a Large Language Model. *IEEE Transactions on Software Engineering* (2026). doi:10.1109/TSE.2026.3682975 arXiv:2506.02954.

[54] Xinyi Wang, Tongxuan Yu, Paolo Arcaini, Tao Yue, and Shaukat Ali. 2022. Mutation-based test generation for quantum programs with multi-objective search. In *GECCO '22: Genetic and Evolutionary Computation Conference, Boston, Massachusetts, USA, July 9 - 13, 2022*, Jonathan E. Fieldsend and Markus Wagner (Eds.). ACM, 1345–1353. doi:10.1145/3512290.3528869