LLMorpheus: Mutation Testing using Large Language Models

Frank Tip
f.tip@northeastern.edu
Northeastern University
Boston, MA, USA

Jonathan Bell
j.bell@northeastern.edu
Northeastern University
Boston, MA, USA

Max Schäfer
max@xbow.com
XBOW
Oxford, UK

ABSTRACT

In mutation testing, the quality of a test suite is evaluated by introducing faults into a program and determining whether the program's tests detect them. Most existing approaches for mutation testing involve the application of a fixed set of mutation operators, e.g., replacing a "+" with a "-", or removing a function's body. However, certain types of real-world bugs cannot easily be simulated by such approaches, limiting their effectiveness. This paper presents a technique for mutation testing where placeholders are introduced at designated locations in a program's source code and where a Large Language Model (LLM) is prompted to ask what they could be replaced with. The technique is implemented in *LLMorpheus*, a mutation testing tool for JavaScript, and evaluated on 13 subject packages, considering several variations on the prompting strategy, and using several LLMs. We find *LLMorpheus* to be capable of producing mutants that resemble existing bugs that cannot be produced by *StrykerJS*, a state-of-the-art mutation testing tool. Moreover, we report on the running time, cost, and number of mutants produced by *LLMorpheus*, demonstrating its practicality.

ACM Reference Format:
Frank Tip, Jonathan Bell, and Max Schäfer. 2025. LLMorpheus: Mutation Testing using Large Language Models. In *Proceedings of XXX*. ACM, New York, NY, USA, 79 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Mutation testing is an approach for evaluating the adequacy of a test suite and is increasingly adopted in industrial settings [48–50]. With mutation testing, an automated tool repeatedly injects a small modification to the system under test and executes the test suite on this mutated code. Mutation testing is premised on the *competent programmer hypothesis*, which posits that most buggy programs are quite close to being correct and that complex faults are *coupled* with simpler faults [18], i.e., a test that is strong enough to detect a simple fault should also be able to detect a more complex one. Hence, mutation analysis tools typically apply a relatively small set of mutation operators: replacing constants, replacing operators, modifying branch conditions, and deleting statements. Studies have shown that, given two test suites for the same system under test, the one that detects more mutants (even using only these limited mutation operators) is likely to also detect more real faults [30, 36].

However, not *all* real faults are coupled to mutants due to the limited set of mutation operators. For example, a fault resulting from calling the wrong method on an object is unlikely to be coupled to a mutant, as state-of-the-art mutation tools do not implement a "change method call" operator. While a far wider range of mutation operators has been explored in the literature [23, 28], state-of-the-practice tools like Pitest [12, 14], Major [29] and Stryker [54] typically do not implement them because of the implementation effort required and, especially, the increased cost of mutation analysis. Each additional mutation operator will result in more mutants that must be run and analyzed, and since each mutant must be evaluated in isolation, this may dramatically increase the time needed for developers that use the tool. Furthermore, some mutation operators might not be worthwhile to run, as noted in documentation from the developer of Pitest: "Although pitest provides a number of other operators, they are not enabled by default as they may provide a poorer experience" [13]. An alternative approach for generating mutants is to use a dataset of real faults to train a machine learning model to learn how to inject mutants [47, 55, 58]. However, the need for developers to train a model for their project is an impediment to adoption of such techniques.

Our approach, *LLMorpheus*, can be viewed as a generalization of rule-based mutation techniques [12, 14, 29, 54] in which the location of mutations is determined using a set of predefined rules, and where an LLM is asked to suggest a diversity of mutations that introduce buggy behavior at those locations. To this end, *LLMorpheus* repeatedly prompts an LLM to inject faults at designated locations into a code fragment using prompts that include: (i) general background on mutation testing (ii) (parts of) a source file in which a single code fragment is replaced with the word "PLACEHOLDER", (iii) the original code fragment that was replaced by the placeholder, and (iv) a request to replace the placeholder with a buggy code fragment that has different behavior than the original code. After discarding syntactically invalid suggestions, we use *StrykerJS*, a state-of-the-art mutation testing tool for JavaScript that we modified to apply the mutations suggested by *LLMorpheus* instead of applying its standard mutators, classify mutants as killed, surviving, or timed out, and generate an interactive web site for inspecting the results.

The effectiveness of our approach hinges on the assumption that LLMs can understand the surrounding context of the code fragment represented by a PLACEHOLDER well enough to suggest syntactically valid and realistic buggy code fragments. To determine whether this assumption holds, we evaluate *LLMorpheus* on 13 subject applications written in JavaScript and TypeScript and measure how many mutants are generated and how they are classified (killed, survived, timed-out) using four "open" LLMs for which the training process is documented (Meta's *codellama-34b-instruct*, *codellama-13b-instruct*, *llama-3.3-70b-instruct* and Mistral's *mixtral-8x7b-instruct*) and one proprietary LLM (OpenAI's *gpt-4o-mini*). We manually examine a subset of the surviving mutants to determine whether they are equivalent to the original source code or if they represent behavioral changes and contrast the results against mutants generated using *StrykerJS*'s standard mutators. The cost of *LLMorpheus* is assessed by measuring its running time and the number of tokens used for prompts and completions. We also report on experiments with alternative prompts that omit parts of the information encoded in default prompts and with different "temperature" settings of an LLM.

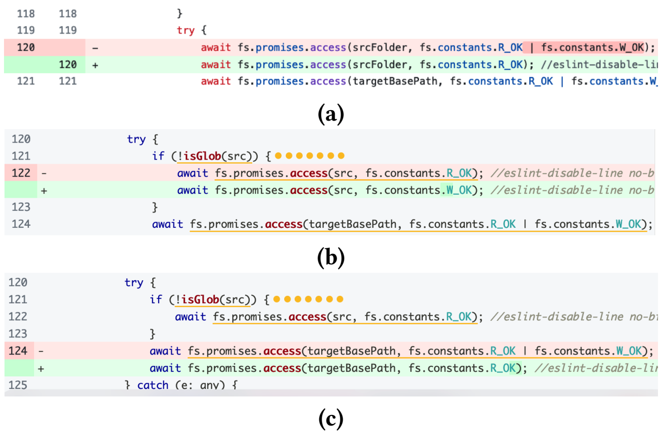
To investigate *LLMorpheus*'s ability to produce mutants that resemble real-world faults, we conducted a detailed case study involving four real-world bugs from the Bugs.js suite [25]. In this case study, we used *LLMorpheus* to mutate the *fixed* version of a program near the location of the fix, executed the program's tests for each of these mutants and compared the test outcomes against those of the buggy version. In each case, *LLMorpheus* produced either a mutant that was identical to a bug that was originally observed, or one that caused the same test failures as a bug that was originally observed. Hence, we find *LLMorpheus* to be capable of generating a diverse set of mutants, some of which resemble real-world bugs that cannot be created by *StrykerJS*' standard mutation operators.

For surviving mutants generated using *codellama-34b-instruct*, we find that the majority (78%) reflect behavioral differences and 20% are equivalent to the original code. Using the *codellama-34b-instruct* and *codellama-13b-instruct* models, results are generally stable at temperature 0.0 when experiments are repeated, but the use of higher temperatures yields more variable results. For *mixtral-8x7b-instruct*, *llama-3.3-70b-instruct*, and *gpt-4o-mini* models, there is already significant variability at temperature 0. The default template generally produces the largest number of mutants and surviving mutants, and removing different fragments of this prompt degrades the results to varying degrees. The *llama-3.3-70b-instruct* and *codellama-34b-instruct* LLMs generally produce the largest number of mutants and surviving mutants, but *LLMorpheus* is still effective when *codellama-13b-instruct*, *mixtral-8x7b-instruct*, and *gpt-4o-mini* are used.

In summary, the contributions of this paper are:

(1) A technique for mutation testing in which placeholders are introduced at designated locations in a program's source code, and where an LLM is prompted to ask what they could be replaced with.
(2) An implementation of this technique in *LLMorpheus*, a practical mutation testing tool for JavaScript.
(3) An empirical evaluation of *LLMorpheus* on 13 subject applications, demonstrating its practicality and comparing it to a standard approach to mutation testing based on mutation operators.

The remainder of this paper is organized as follows. Section 2 presents motivating examples that illustrate the potential of LLM-based mutation techniques to introduce faults resembling real bugs. In Section 3, an overview of our approach is presented. Section 4 presents an evaluation of *LLMorpheus* and Section 5 covers threats to validity. Related work is discussed in Section 6. Lastly, Section 7 presents conclusions and directions for future work.



```
            }
            try {
-               await fs.promises.access(srcFolder, fs.constants.R_OK | fs.constants.W_OK);
+               await fs.promises.access(srcFolder, fs.constants.R_OK); //eslint-disable-li
                await fs.promises.access(targetBasePath, fs.constants.R_OK | fs.constants.W_
```

(a)

```
        try {
            if (!isGlob(src)) {
-               await fs.promises.access(src, fs.constants.R_OK); //eslint-disable-line no-b
+               await fs.promises.access(src, fs.constants.W_OK); //eslint-disable-line no-b
            }
            await fs.promises.access(targetBasePath, fs.constants.R_OK | fs.constants.W_OK);
```

(b)

```
        try {
            if (!isGlob(src)) {
                await fs.promises.access(src, fs.constants.R_OK); //eslint-disable-line no-b
            }
-           await fs.promises.access(targetBasePath, fs.constants.R_OK | fs.constants.W_OK);
+           await fs.promises.access(targetBasePath, fs.constants.R_OK); //eslint-disable-li
        } catch (e: any) {
```

(c)

Figure 1: (a) Fix for a bug reported in issue #36 in `zip-a-folder`. (b) A mutation suggested by *LLMorpheus* at the same line that involves replacing read-access with write-access. (c) A mutation suggested by *LLMorpheus* elsewhere in the same file that mirrors the change made by the developer.

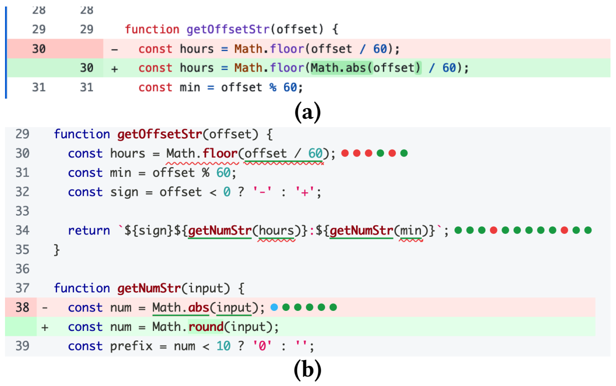
2 BACKGROUND AND MOTIVATION

In this section, we study a few bugs that do not correspond to mutation operators supported by state-of-the-art mutation testing tools but that are similar to mutations *LLMorpheus* could suggest.

Example 1. Zip-a-folder [4] is a library for compressing folders. On January 31, 2022, a user observed that the library required write-access for source-folders unnecessarily and opened issue #36, requesting that this access be removed. The developer applied the fix shown in Figure 1(a) on the same day, which involves replacing a binary bitwise-or expression with one of its operands.

LLMorpheus can suggest mutations that involve *changing or introducing* references to functions, variables, and properties. Figure 1(b) and (c) show two mutations that *LLMorpheus* suggests for this project and that could result in bugs similar to the one described above: part (b) shows a mutation at the same line where the bug was located that involves replacing read-access with write-access, and part (c) shows a mutation at a nearby location that mirrors the change made by the developer.

The state-of-the-art *StrykerJS* tool is unable to suggest either of these mutations because (i) it does not support the mutation of bitwise operator expressions such as `fs.constants.R_OK | fs.constants.W_OK` unless they appear as part of a control-flow predicate, nor (ii) mutations that involve replacing a binary expression with one of its operands. While adding support for mutating bitwise operator expressions would be straightforward, concerns have been expressed that adding more mutation operators to traditional mutation testing tools might result in too many mutants and degraded



(a)

```
    function getOffsetStr(offset) {
-     const hours = Math.floor(offset / 60);
+     const hours = Math.floor(Math.abs(offset) / 60);
      const min = offset % 60;
```

(b)

```
  function getOffsetStr(offset) {
    const hours = Math.floor(offset / 60);
    const min = offset % 60;
    const sign = offset < 0 ? '-' : '+';

    return `${sign}${getNumStr(hours)}:${getNumStr(min)}`;
  }

  function getNumStr(input) {
-   const num = Math.abs(input);
+   const num = Math.round(input);
    const prefix = num < 10 ? '0' : '';
```

Figure 2: (a) Fix for a bug reported in issue #60 in countries-and-timezones. (b) A mutation suggested by *LLMorpheus* elsewhere in the same file.

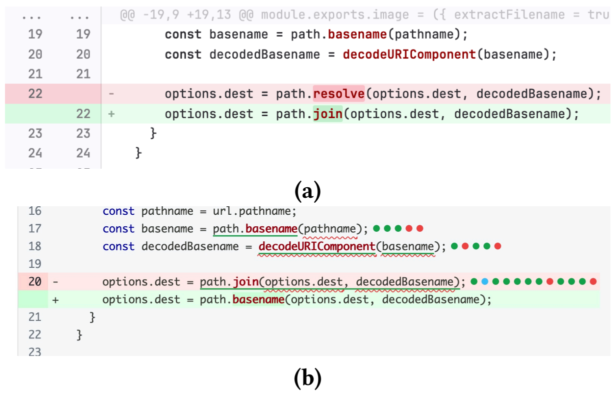
performance [13, 31, 32]. More significantly, *StrykerJS* does not introduce or modify property access expressions and has very limited support for replacing an expression with a different expression[1].

Example 2. Countries-and-timezones [2] is a library for working with countries and timezones. In October 2023, a user reported a bug in function `getOffsetStr`, stating that it produces incorrect results when invoked with negative values. The developer proposed a simple fix that involves inserting a call to `Math.abs` to convert the argument value to a non-negative number, and a variation on this fix was quickly adopted by the developer, as shown in Figure 2(a).

This bug-fix involves the introduction of a function call, so to *introduce* bugs like this one, a mutation testing tool would have to remove function calls or change the function being invoked. *StrykerJS* only supports a very limited set of 20 mutations to function calls[2], such as replacing calls to `String.startsWith` with call to `String.endsWith` and removing a call to `Array.slice`. While one could extend *StrykerJS* with a mutator that removes calls to `Math.abs`, many other function calls could be handled similarly, and adding mutators for all of them would yield an overwhelmingly large number of mutants. Many such candidate functions would not be good choices for mutation, either because the function in question is not a function that a developer inadvertently might have selected, or because it would lead to syntactically invalid code.

LLMorpheus suggests mutations that involve introducing and replacing function calls. Figure 2(b) shows a mutation that *LLMorpheus* suggested elsewhere in the same source file that involves replacing a call to `Math.abs` with a call to `Math.round`, which could, in principle, introduce a bug like the one in Figure 2(a). Moreover, since LLMs are trained to generate code that resembles code written by developers, it is likely that the mutants produced by *LLMorpheus* involve the use of functions that a developer might have chosen.

Example 3. image-downloader is a module for downloading images. In February 2022, a user opened issue #27, entitled "If the



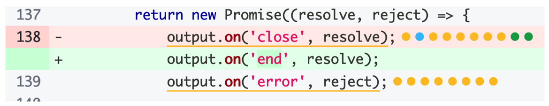
(a)

```
@@ -19,9 +19,13 @@ module.exports.image = ({ extractFilename = tru
          const basename = path.basename(pathname);
          const decodedBasename = decodeURIComponent(basename);

-         options.dest = path.resolve(options.dest, decodedBasename);
+         options.dest = path.join(options.dest, decodedBasename);
        }
      }
```

(b)

```
        const pathname = url.pathname;
        const basename = path.basename(pathname);
        const decodedBasename = decodeURIComponent(basename);

-       options.dest = path.join(options.dest, decodedBasename);
+       options.dest = path.basename(options.dest, decodedBasename);
      }
    }
```

Figure 3: (a) Fix for a bug reported in issue #27 in image-downloader. (b) A mutation suggested by *LLMorpheus* at the same location that similarly involves calling a different function.



```
        return new Promise((resolve, reject) => {
-           output.on('close', resolve);
+           output.on('end', resolve);
            output.on('error', reject);
```

Figure 4: A mutation suggested by *LLMorpheus* that involves associating an event listener with the `end` event instead of with the `close` event.

directory name in dest: contains a dot "." then the download fails.", providing an example illustrating the problem. The developers soon responded with a fix, shown in Figure 3(a), that involves replacing a call to `path.resolve` with a call to `path.join`. While *LLMorpheus* does not produce a mutant that re-introduces this bug exactly, it does produce several at the same location[3] that similarly replace the invoked function, including the one shown in Figure 3(b). As mentioned, *StrykerJS* has very limited support for mutations that involve calling different functions and so it cannot suggest mutations like the one shown in Figure 3(b).

Example 4. Figure 4 shows another mutant produced by *LLMorpheus* for *zip-a-folder*. Here, the mutation involves changing the name of the event with which an event listener is associated. Such errors often cause "dead listeners", i.e., situations where an event handler is never executed because it is associated with the wrong event. Dead listeners are quite common in JavaScript, where the use of string values to identify events precludes static checking, and previous research has focused on static analysis [43] and statistical methods [5] for detecting such errors.

Discussion. The above examples illustrate just a few of the kinds of mutations that *LLMorpheus* may produce. Other mutations that it may suggest include: replacing a reference to a variable with a reference to a different variable, adding or removing arguments in function calls, and modifying object literals by adding or removing property-value pairs.

[1]In particular, *StrykerJS* only replaces control-flow predicates in **if**-statements and loops with boolean constants, string literals with the value "Stryker␣was␣here", and object literals with an empty object literal.

[2]See https://stryker-mutator.io/docs/mutation-testing-elements/supported-mutators/.

[3]The line numbers have shifted slightly as the code has evolved since the bug report.

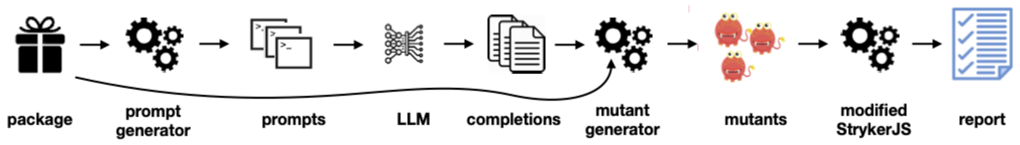


Figure 5: Overview of approach.

In practice, the number of such mutations is effectively infinite, so an approach based on exhaustively applying a fixed set of mutation operators is unlikely to be practical. *LLMorpheus*' LLM-based approach leverages the collective wisdom of programmers who wrote the code on which the LLM was trained to develop mutations. As a result, suggested changes are likely to refer only to variables and functions that are in scope, and are likely to be type-correct.

3 APPROACH

LLMorpheus is capable of producing interesting mutants without requiring any training on a subject project, which is a key distinction compared to existing work that builds models of real bugs to generate mutants [47, 55, 58]. This is accomplished by querying an LLM with a prompt that includes part of an application's source code in which a code fragment is replaced with the text "<PLACEHOLDER>". Additional information provided in the prompt includes: (i) general background on mutation testing, (ii) the code fragment that was originally present at the placeholder's location, (iii) a request to apply mutation testing to the code by replacing the placeholder with a buggy code fragment, and (iv) suggestions *how* the code could be mutated. The LLM is asked to provide three possible replacements for the placeholder, each accompanied by an explanation how the mutation would change program behavior.

Figure 5 presents a high-level overview of our approach, which involves three components that work in concert: the *prompt generator*, the *mutant generator*, and a version of the *StrykerJS* mutation testing tool that has been modified to apply the mutants created by *LLMorpheus*[4] We now discuss each of these components.

Prompt generator. This component takes as input a package and generates a set of prompts. The prompt generator parses the source files and identifies locations where mutations will be introduced. For ease of reference during prompting, the source code fragment corresponding to each location is replaced with the text "<PLACEHOLDER>". *LLMorpheus* considers the following locations as candidates for mutation: (i) conditions of **if**, **switch**, **while**, and **do-while** statements, (ii) initializers, updaters, and entire headers of loop statements, and (iii) receiver, arguments, and entire sequence of arguments for function calls. For each such location, a separate prompt is created. Figure 6 illustrates where placeholders are introduced into the source code.

The LLM is then given a prompt that is created by instantiating the template shown in Figure 7(a), by replacing {{{ code }}} with the original source code in which a placeholder has been inserted, and {{{ orig }}} with the code fragment that was replaced by the placeholder. Figure 7(b) shows the system prompt given to the LLM, which provides background on the role the LLM is expected to play

if (x === y){ ... }	**if** (<PLACEHOLDER>){ ... }
switch (x === y){ ... }	**switch** (<PLACEHOLDER>){ ... }
while (x){ ... }	**while** (<PLACEHOLDER>){ ... }
do { ... } **while** (x)}	**do** { ... } **while** (<PLACEHOLDER>)

for (**let** i=0; i < x; i++){
...
}
	for (<PLACEHOLDER>; i < x; i++){ ... }
	for (**let** i=0; <PLACEHOLDER>; i++){ ... }
	for (**let** i=0; i < x; <PLACEHOLDER>){ ... }
	for (<PLACEHOLDER>){ ... }

for (o **in** obj){ ... }	**for** (<PLACEHOLDER> **in** obj){ ... }
	for (o **in** <PLACEHOLDER>){ ... }
	for (<PLACEHOLDER>){ ... }

for (o **of** obj){ ... }	**for** (<PLACEHOLDER> **of** obj){ ... }
	for (o **of** <PLACEHOLDER>){ ... }
	for (<PLACEHOLDER>){ ... }

a.m(x,y)	<PLACEHOLDER>(x,y)
	a.m(<PLACEHOLDER>,y)
	a.m(x,<PLACEHOLDER>)
	a.m(<PLACEHOLDER>)

Figure 6: Illustration of the insertion of placeholders to direct the LLM at source locations that need to be mutated.

in the conversation as a mutation testing expert. As can be seen in Figure 7(a), the prompt provides instructions for applying mutation testing to the specific source code at hand and details the specific format to which the completion should conform. Specifically, we require that the proposed mutants be provided inside "fenced code blocks" (i.e., code blocks surrounded by three backquote characters).

Mutant generator. This component takes the completions received from the LLM and extracts candidate mutants from the instantiated template by matching a regular expression against the completion to find the fenced code blocks. Candidate mutants identical to the original source code fragment or identical to previously generated mutants are discarded. The candidate mutants are then parsed to check if they are syntactically valid and discarded if this is not the case. The resulting mutants are written to a file mutants.json that is read by a customized version of *StrykerJS* that is described below. The mutant generator also saves all experimental data to files, including the generated prompts, completions received from the LLM, and the configuration options that were used (e.g., the LLM's temperature setting).

Custom version of StrykerJS. We modified *StrykerJS* to give it an option --usePrecomputed that, if selected, directs it to read its set of mutations from a file mutants.json instead. *StrykerJS* then executes all mutants and determines for each mutant whether it causes tests to fail or time out. When this analysis is complete, *StrykerJS* generates a report as an interactive web page allowing users to inspect the generated mutants. The previously shown Figures 1–4 show screenshots of our custom version of *StrykerJS*.

Pragmatics. While *LLMorpheus* implements a conceptually straightforward technique, considerable engineering effort was required

[4]In particular, we use *StrykerJS*' to (i) determine the impact of each mutant on an application's tests and classify it as "killed", "survived", or "timed-out" and (ii) generate an interactive web page for inspecting results.

~~~
Your task is to apply mutation testing to the following
code:
```
{{{code}}}
```

by replacing the PLACEHOLDER with a buggy code fragment
that has different behavior than the original code fragment,
which was:
```
{{{orig}}}
```
Please consider changes such as using different operators,
changing constants, referring to different variables, object
properties, functions, or methods.

Provide three answers as fenced code blocks containing a
single line of code, using the following template:

Option 1: The PLACEHOLDER can be replaced with:
```
<code fragment>
```
This would result in different behavior because
<brief explanation>.

Option 2: The PLACEHOLDER can be replaced with:
```
<code fragment>
```
This would result in different behavior because
<brief explanation>.

Option 3: The PLACEHOLDER can be replaced with:
```
<code fragment>
```
This would result in different behavior because
<brief explanation>.

Please conclude your response with "DONE."
~~~

(a)

```
You are an expert in mutation testing. Your job is to
make small changes to a project's code in order to find
weaknesses in its test suite. If none of the tests fail
after you make a change, that indicates that the tests
may not be as effective as the developers might have
hoped, and provide them with a starting point for
improving their test suite.
```

(b)

Figure 7: Prompt template (a) and system prompt (b) used by *LLMorpheus.*

to make it a practical tool. We use BabelJS [1] for parsing source code to identify locations where placeholders should be inserted, and to check syntactic validity of candidate mutants. Handlebars [3] is used for instantiating prompt templates. *StrykerJS* expects mutants to correspond to a single AST node, so for mutants that do not correspond exactly to a single AST node (e.g., loop headers and sequences of arguments passed in function calls), it is necessary to expand the mutation to the nearest enclosing AST node, for which we also rely on BabelJS.

LLMorpheus has command-line arguments for specifying the prompt template and system template to be used. Furthermore, it enables users to specify a number of LLM-specific parameters, such as the maximum length of completions that should be generated, the sampling temperature[5], and number of lines of source code that should be included in prompts (by default, this is limited to 200 lines surrounding the location of the placeholder). Since many LLM installations have limited capacity or explicit rate limits, *LLMorpheus* provides two command-line options to work with such LLMs: `--rateLimit <N>` ensures that least N milliseconds will have elapsed between successive prompts and `--nrAttempts <N>` will try the same prompt up to N times if a 429 error occurs.

One possible concern with our approach is that *LLMorpheus* relies on a fixed set of locations where it introduces placeholders. The current placeholder scheme aims to strike a balance between creating a number of mutants that is practical and having mutants that are likely to result in different control flow or data flow and that are likely to be different than what can be achieved using mutation operators. It would be straightforward to modify *LLMorpheus* to use a different placeholder scheme. That said, the examples in Section 2 show that mutants produced by *LLMorpheus* (using its current placeholder scheme) involve changing references to variables, properties, and functions that cannot be produced using Stryker's mutation operators and that correspond to real-world bugs.

An open-source release of *LLMorpheus* can be found at https://github.com/neu-se/llmorpheus and the customized version of *StrykerJS* that we used for classifying mutants can be found at https://github.com/neu-se/stryker-js.

4 EVALUATION

4.1 Research Questions

This evaluation aims to answer the following research questions:

RQ1 How many mutants does *LLMorpheus* create?
RQ2 How many of the surviving mutants produced by *LLMorpheus* are equivalent mutants?
RQ3 What is the effect of using different temperature settings?
RQ4 What is the effect of variations in the prompting strategy used by *LLMorpheus*?
RQ5 How does the effectiveness of *LLMorpheus* depend on the LLM that is being used?
RQ6 What is the cost of running *LLMorpheus*?
RQ7 Is *LLMorpheus* capable of producing mutants that resemble existing bugs?

4.2 Experimental Setup

Selecting subject applications. Our goal is to evaluate *LLMorpheus* on real-world JavaScript packages that have test suites. Moreover, we want to compare the mutants generated by*LLMorpheus* to those generated using traditional mutation testing techniques, so we decided to focus on projects for which the state-of-the-art StrykerJS mutation testing tool [54] could be applied successfully. As a starting point for benchmark selection, we considered the 25 subject applications that were used to evaluate TestPilot [51], a recent LLM-based unit test generation tool. These applications are written in JavaScript or TypeScript, cover a range of different domains, and have test suites that can be executed successfully.

Of these 25 subject applications, 10 could not be used because StrykerJS does not work on them, either because its dependences conflict with those of the subject application itself[6], or because

[5]The sampling temperature is a parameter between 0 and 2 that controls the randomness of the completions generated by the LLM. Roughly speaking, the higher the temperature the more diverse the completions. At temperature zero, the LLM will always generate the most likely completion, which increases the chance that the same prompt will result in the same completion.

[6]Running StrykerJS on an application requires installing it locally among the subject project libraries. Stryker itself depends on various other packages that also need to be

application	description	weekly	#LOC	#tests	coverage		*StrykerJS*					
		downloads			stmt	branch	#mutants	#killed	#survived	#timeout	mut. score	time (sec)
Complex.js	complex numbers	671K	1,425	216	71.82%	67.54%	1,302	763	539	0	58.60	405.08
countries-and-timezones	accessing countries and timezones data	152K	165	58	100%	92.55%	140	134	6	0	95.71	142.37
crawler-url-parser	URL parser for crawling	495	209	185	96.39%	92.5%	226	143	83	0	63.27	433.53
delta	Format for representing rich text documents and changes	1.76M	806	180	98.99%	95.89%	834	686	88	60	89.45	2747.04
image-downloader	downloading image to disk from a given URL	17.75K	64	11	100%	93.75%	43	28	11	4	74.42	284.20
node-dirty	key value store with append-only disk log	5,604	207	37	83.01%	71.15%	160	78	56	26	65.00	215.93
node-geo-point	calculations involving geographical coordinates	5,618	406	10	85.36%	70.58%	158	98	60	0	62.03	357.70
node-jsonfile	reading/writing JSON files	57.7M	102	43	97.87%	94.11%	61	31	5	25	91.80	188.91
plural	plural forms of nouns	2,271	103	14	95.38%	72.72%	180	143	37	0	79.44	53.66
pull-stream	pipeable pull-stream	57.8K	602	364	90.96%	80.84%	474	318	116	40	75.53	694.33
q	promises	10.1M	2,111	243	89.5%	70.92%	1,058	68	927	63	12.38	7,075.02
spacl-core	path-based access control	3	377	38	100%	100%	259	239	20	0	92.28	1,053.16
zip-a-folder	zip/tar utility	60.1K	156	22	100%	96.87%	74	38	8	28	89.19	513.27

Table 1: Subject applications used to evaluate *LLMorpheus*.

it crashes. On one package, *simple-statistics*, StrykerJS requires approximately 10 hours of running time, which makes using it impractical. We excluded another package, *fs-extra*, a utility library for accessing the file system, because we observed that mutating this application poses a significant security risk, as the mutated code was corrupting our local file system. This left us with 13 subject applications for which Table 1 provides key characteristics. The first set of columns in the table show, from left to right, the name of the package, a short description of its functionality, the number of weekly downloads according to npmjs.com, the number of lines of source code, the number of tests, and the statement and branch coverage achieved by those tests, respectively. The second set of columns shows the results of running *StrykerJS* on the applications: the total set of mutants, the number of mutants that were killed, survived, and that timed out, the *mutation score*[7] reported by *StrykerJS*, and the time required to run *StrykerJS*, respectively.

LLM selection. RQ5 explores how the effectiveness of the proposed technique depends on the LLM being used. We use Meta's *codellama-34b-instruct* model for our main experiments. In addition, we evaluate the technique with Meta's *codellama-13b-instruct* and *llama-3.3-70b-instruct* models, with Mistral's *mixtral-8x7b-instruct* model, and with OpenAI's *gpt-4o-mini* model. The *codellama* models are specifically trained for tasks involving code. *llama-3.3-70b-instruct* is a newer and larger model from Meta that supercedes the smaller, specialized *codellama* models. *mixtral-8x7b-instruct* is a state-of-the-art general-purpose "mixture-of-experts" LLM developed by Mistral. *gpt-4o-mini* is a smaller, faster, and lower-cost variant of OpenAI's popular *gpt-4o* model The *codellama-34b-instruct*, *codellama-13b-instruct*, *llama-3.3-70b-instruct*, and *mixtral-8x7b-instruct* LLMs are "open" in the sense that their training process is documented. We relied on several commercial LLM service providers (https://octo.ai, https://openai.com, and https://openrouter.ai) for the experiments reported on in this paper.

LLM Temperature settings. LLMs have a temperature parameter that reflects the amount of randomness or creativity in their completions. For a task such as mutation testing, randomness and creativity may determine whether generated mutants are killed or survive. Therefore, we conduct experiments using several temperature settings.

Similarity to real-world bugs. Previous work on evaluating mutation testing techniques has focused on "coupling" to determine whether mutants resemble real-world bugs [22, 30, 36]. This involves determining whether a test suite that detects particular mutants also detects particular real faults, and requires a curated dataset of isolated faults. While there is a wealth of such datasets constructed from open-source projects written in Java, we found only one JavaScript dataset, the Bugs.js suite [25]. Unfortunately, we found that most of these subjects could not be used at all due to their reliance on outdated versions of various libraries and because of their incompatibility with modern Node.js versions that *StrykerJS* requires, causing them to be incompatible with *LLMorpheus*. These projects also have flaky tests[8], making it particularly challenging to perform mutation analysis [53]. We, therefore, opted to conduct a case study involving four real-world bugs from the Bugs.js suite that we could reproduce reliably.

4.3 RQ1: How many mutants does *LLMorpheus* create?

To answer this question, we ran *LLMorpheus* on the projects listed in Table 1 using the *codellama-34b-instruct* LLM at temperature 0.0 and the prompt templates shown in Figure 7. The results, shown in Table 2, show that *LLMorpheus* produces between 42 and 1,051 prompts for these projects. The next 4 columns in the table show the number of "candidate mutants", i.e., code fragments obtained by replacing placeholders with code fragments suggested by the LLM. The subcolumn labeled "candidates" shows the total number

installed, and these packages may conflict with packages that the subject application itself depends upon.

[7]The mutation score aims to provide a measure of the quality of a test suite by calculating 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/.

[8]See https://github.com/BugsJS/bug-dataset/issues/11.

application	**#prompts**	*LLMorpheus*									*StrykerJS*				
		#candidates	**#invalid**	**#identical**	**#duplicate**	**#mutants**	**#killed**	**#survived**	**#timeout**	**mut. score**	**#mutants**	**#killed**	**#survived**	**#timeout**	**mut. score**
Complex.js	490	1,451	194	13	45	1,199	725	473	1	60.55	1,302	763	539	0	58.60
countries-and-timezones	106	318	89	0	12	217	188	29	0	86.64	140	134	6	0	95.71
crawler-url-parser	176	521	205	14	17	285	157	128	0	55.09	226	143	83	0	63.27
delta	462	1,367	565	10	25	767	634	101	32	86.83	834	686	88	60	89.45
image-downloader	42	124	33	2	0	89	72	17	0	80.90	43	28	11	4	74.42
node-dirty	154	450	153	15	7	275	163	100	12	63.64	160	78	56	26	65.00
node-geo-point	140	408	93	0	13	302	223	79	0	73.84	158	98	60	0	62.03
node-jsonfile	68	199	42	3	0	154	49	48	57	68.83	61	31	5	25	91.80
plural	153	442	101	42	18	281	205	75	1	73.31	180	143	37	0	79.44
pull-stream	351	1,028	238	12	9	769	441	271	57	64.76	474	318	116	40	75.53
q	1,051	3,121	1,000	34	52	2,035	158	1,792	85	11.94	1,058	68	927	63	12.38
spacl-core	134	395	140	10	6	239	199	39	1	83.68	259	239	20	0	92.28
zip-a-folder	49	143	41	1	1	100	23	3	74	97.00	74	38	8	28	89.19
Total	3,376	9,967	2,894	156	205	6,712	3,237	3,155	320	—	4,969	2,767	1,956	246	—

Table 2: Results from LLMorpheus experiment (run #312). Model: *codellama-34b-instruct*, temperature: 0.0, maxTokens: 250, template: *template-full.hb*, systemPrompt: *SystemPrompt-MutationTestingExpert.txt*.

	temp. 0.0 (run #312)				**temp. 0.25 (run #348)**				**temp. 0.50 (run #318)**				**temp. 1.0 (run #341)**			
	#mutants	**#killed**	**#survived**	**#timeout**	**#mutants**	**#killed**	**#survived**	**#timeout**	**#mutants**	**#killed**	**#survived**	**#timeout**	**#mutants**	**#killed**	**#survived**	**#timeout**
Complex.js	1,199	725	473	1	1,197	730	466	1	1,191	739	452	0	1,028	648	379	1
countries-and-timezones	217	188	29	0	219	181	38	0	224	194	30	0	186	156	30	0
crawler-url-parser	285	157	128	0	260	167	93	0	298	166	108	24	278	202	76	0
delta	767	634	101	32	781	642	111	28	769	642	93	34	698	583	83	32
image-downloader	89	72	17	0	86	71	15	0	89	68	21	0	75	53	22	0
node-dirty	275	163	100	12	279	175	93	11	277	158	107	12	246	150	84	12
node-geo-point	302	223	79	0	293	225	68	0	302	230	72	0	273	213	60	0
node-jsonfile	154	49	48	57	151	52	41	58	153	51	43	59	132	50	22	60
plural	281	205	75	1	273	208	63	2	289	219	69	1	299	229	69	1
pull-stream	769	441	271	57	779	452	270	57	796	465	278	53	743	461	235	47
q	2,035	158	1,792	85	2,050	153	1,813	84	2,073	163	1,823	87	1,899	147	1,671	81
spacl-core	239	199	39	1	223	187	36	0	250	210	39	1	218	180	38	0
zip-a-folder	100	23	3	74	97	24	4	69	87	48	33	6	96	54	38	4

Table 3: Number of mutants generated using the *codellama-34b-instruct* LLM at temperatures 0.0, 0.25, 0.5, and 1.0 (showing one run of each)

of candidate mutants, the subcolumn labeled "invalid" shows the number of candidate mutants that were found to be syntactically invalid, the subcolumn labeled "identical" shows the number of candidate mutants that were found to be identical to the original code, and the subcolumn labeled "duplicate" shows the number of candidate mutants that were found to be duplicated. From this data, it can be inferred that, on average, 29.0% (2,894/9,967) of candidate mutants are discarded because they are syntactically invalid, 1.6% (156/9,967) are discarded because they are identical to the original code, and 2.1% (205/9,967) are discarded because they are duplicates. This suggests that LLMs generally do not have too much trouble with generating syntactically correct code, which is consistent with recent findings by others [38, 51]. The next column, labeled "mutants", shows the number of mutants that remain after discarding the useless candidate mutants. Here, the reader can see that between 89 and 2,035 mutants are generated for the subject packages. Of these mutants, between 23 and 725 are killed, between 3 and 1,792 survive, and between 0 and 85 time out. Aggregating the results over all projects, it can be seen that 48.2% (3,237/6,712) of all mutants are killed, 47.0% (3,155/6,712) of all mutants survive, and 4.8% (320/6,712) of all mutants time out.

The table also shows the *mutation score*[9] as reported by StrykerJS, which aims to provide a measure of the quality of a test suite by calculating the fraction of the total number of mutants that are detected (i.e., killed or timed out).

To facilitate a quantitative comparison with *StrykerJS*, the last 5 columns in Table 2 repeat the results of running *StrykerJS* on the subject applications from Table 1. From this data, it can be seen that—in the aggregate for the 13 projects under consideration—*LLMorpheus* produces 3,155 surviving mutants whereas *StrykerJS* produces 1,956 surviving mutants. However, it should be noted that the difference in the number of mutants and surviving mutants varies significantly between subject applications. For example, for *Complex.js* *StrykerJS* produces more mutants (1,302 vs. 1,199) than *LLMorpheus*, of which more survive (539 vs. 473). On the other hand, for *q*, the situation is reversed with *StrykerJS* producing fewer mutants (1,058 vs. 2,035) and fewer surviving mutants (927 vs.1,792) than *LLMorpheus*. We conjecture that such differences are due to the subject programs having different characteristics, which makes them amenable to different types of mutations. Here, *Complex.js* makes heavy use of arithmetic operators to implement mathematical operations on complex numbers, and such operators are prime candidates for *StrykerJS*'s standard mutation operators.

[9] See https://stryker-mutator.io/docs/General/faq/.

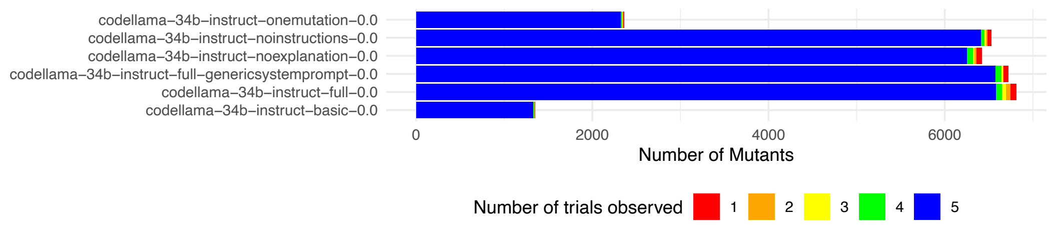


Figure 8: Stability of mutants generated by LLMorpheus with *codellama-34b-instruct* at temperature 0.0. For each replacement generated at each position, we count the number of trials (of 5 total) where that replacement was generated.

	LLMorpheus			Stryker.js		
Project	**equiv**	**not equiv**	**unknown**	**equiv**	**not equiv**	**unknown**
Complex.js	6	44	0	0	50	0
countries-and-timezones	17	11	1	0	6	0
crawler-url-parser	16	33	1	0	47	3
delta	2	48	0	2	48	0
image-downloader	4	12	1	0	4	4
node-dirty	13	36	1	0	46	4
node-geo-point	1	47	2	1	41	8
node-jsonfile	7	25	0	0	3	0
plural	16	34	0	0	36	1
pull-stream	3	44	3	0	48	2
q	2	48	0	0	48	2
spacl-core	18	21	0	2	12	6
zip-a-folder	0	0	0	0	6	0
Total	105	403	9	5	395	30

Table 4: Number of equivalent surviving mutants generated by *LLMorpheus* and *StrykerJS*.

Moreover, *q* makes heavy use of method calls, which are targeted by *LLMorpheus*'s placeholder-based strategy but much less so by *StrykerJS*'s mutation operators.

LLMs are nondeterministic, even at temperature 0, so a subsequent experiment may produce results that differ from those shown in Table 2. To determine to what extent this is the case, we repeated the same experiment 4 more times and measured how often the same mutants occur. We found that, at temperature 0.0, the results of *LLMorpheus* are generally stable across runs, with between 89.29% and 98.89% of all mutants being observed in all 5 experiments[10]. Figure 8 visualizes the stability of *LLMorpheus* when varying the prompt settings, and our supplemental materials include results across all settings of all models.

Using *codellama-34b-instruct* at temperature 0, *LLMorpheus* generates between 89 and 2,035 mutants, of which between 3 and 1,792 survive. These results are stable across experiments, with between 89.29% and 98.89% of all mutants being observed in all five experiments.

4.4 RQ2: How many of the surviving mutants are equivalent mutants?

One of the key challenges in mutation testing is the phenomenon of *equivalent mutants*: mutants that have equivalent behavior as the original code [18]. Mutants produced by *LLMorpheus* may involve arbitrary code changes, so it is entirely possible for the LLM to suggest code that is effectively a refactored version of the code that was originally present. Determining mutant equivalence requires a deep understanding of the program's intended behavior, as a mutant might change the behavior of the program but within bounds that are valid to the program's specification. To determine to what extent surviving mutants produced by *LLMorpheus* are equivalent, we conducted a study in which two authors manually examined 50 surviving mutants[11] in each project and classified each mutant as "equivalent", "not equivalent", or "unknown". After independently coding each sampled mutant, the two authors met, discussed disagreement, and came to a consensus on all mutants. We define these categories as follows.

We labeled a mutant as *equivalent* if we could determine that the change could not cause *any* observable difference in behavior. For example, mutants that added extra parameters to methods (beyond those accepted by the receiver method) are trivially equivalent, as they are discarded by the runtime. Other mutants are far from trivial to evaluate, and we manually wrote test code to attempt to discern the impact of, e.g., changing a condition from `if (! handler)` to `if (handler === undefined)`. Such a mutant is equivalent if `handler` is never any other "falsy" value (e.g., `null`, `false`, `NaN`, `0`, or the empty string `''`).

We labeled a mutant as *not equivalent* if it produced a change that could be observed as a behavioral change to a client of the library. For example, a mutant in the statement `const hours=Math.floor (totalMinutes /60)` that changes the call from `Math.floor` to `Math.round` will result in the value of `hours` being incorrect. Of course, if `hours` is never used (or it doesn't matter that it is off-by-one), then the mutant could still be equivalent. Hence, we also found it necessary to trace through code to determine that some mutants were not equivalent.

In some cases, the mutant resulted in a change to the behavior of the program, but it was truly impossible to determine whether or not this change impacted the correctness of the program as intended by its developers. For example, a mutant that redirects a log statement from `console . error` to `console . warn` may or may not represent a mutant that developers find to deviate from the application's intended behavior. Similarly, mutants that change the contents of error messages may or may not represent an equivalent mutant. We labeled all such cases *unknown.*

The results are shown in Table 4. Of the 517 mutants under consideration, the majority 403 (78%) are "not equivalent", 105 (20%) are "equivalent", and remaining 9 (2%) are classified as "unknown". To place these findings in perspective, we applied the same manual classification to up to 50 surviving mutants produced by *StrykerJS* and

[10]All experimental data associated with this experiment and the other experiments are included with this submission as supplemental materials.

[11]For projects with fewer than 50 surviving mutants, we used as many as were available.

found that of 430 surviving mutants, 395 (92%) are “not equivalent”, 5 (1%) are “equivalent”, and 30 (7%) are classified as “unknown”.

We further examined the 105 equivalent mutants, and observed a number of common patterns, including: (i) checking for `null`-ness or `undefined`-ness in different ways (e.g., replacing `x != null` with `!x` or vice versa), (ii) refactoring of calls to the `String.substring` method with one of its near-equivalent counterparts `String.substr` and `String.slice`, (iii) adding modifiers such as `/g` or `/m` to a regular expression in cases where this does not have any effect, (iv) calls to the `Array.slice` method in cases where this does not have any effect, and (v) calling functions with more arguments than are declared. For the 105 equivalent mutants under consideration, approximately 40% fall into one of these categories. We expect that most of these equivalent mutants can be filtered out using an AST-based static analysis. However, further investigation is needed because some mutants that cause behavioral differences are syntactically similar to these patterns. This means that any pattern-matching-based approach should consider the context in which the mutation occurs to determine whether a mutant is likely to be equivalent. Section 7 will discuss future work that aims to reduce the number of equivalent mutants.

The majority (78%) of the surviving mutants produced by *LLMorpheus* are not equivalent to the original code fragments they replace. *LLMorpheus* produces significantly more “equivalent” mutants than *StrykerJS*. However, the number of “not-equivalent” mutants exceeds the number of equivalent mutants by more than a factor of three, and preliminary analysis reveals good potential for future work on automatically filtering out equivalent mutants using static analysis.

4.5 RQ3: What is the effect of different temperature settings?

To explore the impact of an LLM’s temperature setting, we repeated the experiment with the *codellama-34b-instruct* LLM using temperatures 0.25, 0.50, and 1.0. The results of these experiments are summarized in Table 3. As can be seen from the table, the total number of mutants and the number of surviving mutants at temperatures 0.0, 0.25, and 0.50 are generally fairly similar. However, at temperature 1.0, both the total number of mutants and the number of surviving mutants decline noticeably compared to the results for temperature 0.0. Inspection of the results revealed that this is partly because more of the generated mutants are syntactically invalid.

A secondary question is how temperature affects the variability of results. To answer this question, we repeated the experiment 5 times at each temperature and measured how many distinct mutants occur and how many mutants occur in all five runs. We found that, at higher temperatures, the number of distinct mutants increases rapidly and that the number of mutants common to all runs decreases accordingly. For example, for *Complex.js*, *LLMorpheus* generates 1,217 distinct mutants at temperature 0 of which 1,181 (97.04%) are common to all five runs. At temperature 0.25, the number number of distinct mutants increases to 2,354, of which 447 (18.99%) are common to all five runs. At temperature 0.5, there are 3,196 distinct mutants of which 205 (6.41%) are common to all runs. At temperature 1.0, there are 4,200 distinct mutants, of which 17 (0.4%) are common to all runs, meaning that, effectively, at temperature 1.0 each run produces completely different mutants. The results for the other subject applications are similar. The supplemental materials associated with this paper include an analysis showing the overall variability in mutants killed and survived across each of the five runs.

LLMorpheus generally produces similar numbers of mutants at temperatures ≤ 0.5, of which a similar number survives. At temperature 1.0, the number of generated and surviving mutants decline noticeably because more candidate mutants are syntactically invalid. The variability of results is inversely dependent on the temperature, with mostly the same mutants being produced at temperature 0, and mostly different mutants at temperature 1 in different runs.

4.6 RQ4: What is the effect of variations in the prompting strategy used by *LLMorpheus*?

Thus far, we have evaluated the effectiveness of the prompt template of Figure 7(a) (henceforth referred to as `full`) by measuring how many mutants are generated and classifying them as “killed”, “survived”, or “timed-out” (see Table 2). To determine what the effect is of each component of this prompt, we experimented with the following variations[12]:

onemutation. This variant requests just one replacement of the placeholder instead of three possible replacements.

noexplanation. This variant omits the phrase “`This would result in different behavior because <brief explanation>.`”.

noinstructions. This variant omits the phrase “`Please consider changes such as using different operators, changing constants, referring to different variables, object properties, functions, or methods.`”

genericsystemprompt. In this variant, we replace the system prompt of Figure 7(b) with a generic message “`You are a programming assistant. You are expected to be concise and precise and avoid any unnecessary examples, tests, and verbosity.`”

basic. This minimal template only asks the LLM to provide a code fragment that the placeholder can be replaced with, without any additional context.

Table 5 shows, for each template, the total number of mutants, and the number that were killed, survived, and timed out, respectively. From these results, it can be seen that:

- *full* and *genericsystemprompt* produced the most mutants and performed similarly, demonstrating that the use of a specialized system prompt has minimal impact,
- *noexplanation* and *noinstructions* produce only slightly fewer mutants and surviving mutants than *full* and *genericsystemprompt*, so including instructions or requesting explanations for suggested mutations has limited impact,
- using *onemutation* dramatically reduces the number of mutants from 6,712 to 2,333, demonstrating that it is helpful to request multiple suggestions, and
- using *basic* reduces the number of mutants to 1,326, suggesting that additional context in prompts is helpful.

We separately analyzed the variability of these results (Table 5 presents the results from a single trial), and found the number of

[12]All prompt templates are included with the supplemental materials.

	full				**onemutation**				**noexplanation**				**noinstructions**				**genericsystemprompt**				**basic**			
	(run #312)				(run #365)				(run #372)				(run #378)				(run #384)				(run #390)			
	#mutants	**#killed**	**#survived**	**#timeout**	**#mutants**	**#killed**	**#survived**	**#timeout**	**#mutants**	**#killed**	**#survived**	**#timeout**	**#mutants**	**#killed**	**#survived**	**#timeout**	**#mutants**	**#killed**	**#survived**	**#timeout**	**#mutants**	**#killed**	**#survived**	**#timeout**
Complex.js	1,199	725	473	1	406	245	161	0	1,125	676	448	1	1,137	696	440	1	1,199	740	458	1	185	120	65	0
countries-and-timezones	217	188	29	0	79	65	14	0	211	183	28	0	218	174	44	0	217	191	26	0	48	44	4	0
crawler-url-parser	285	157	128	0	86	50	36	0	239	140	99	0	246	134	112	0	246	143	103	0	67	49	18	0
delta	767	634	101	32	266	221	37	8	734	598	110	26	759	612	115	32	790	659	99	32	201	167	28	6
image-downloader	89	72	17	0	34	26	8	0	77	62	15	0	84	69	15	0	88	72	16	0	10	7	3	0
node-dirty	275	163	100	12	99	55	41	3	258	146	99	13	260	146	103	11	277	162	104	11	44	24	18	2
node-geo-point	302	223	79	0	104	74	30	0	297	216	81	0	306	230	76	0	305	229	76	0	62	54	8	0
node-jsonfile	154	49	48	57	57	18	18	21	152	54	45	53	148	45	51	52	150	49	49	52	22	11	3	8
plural	281	205	75	1	100	70	30	0	273	198	74	1	261	189	71	1	272	209	62	1	92	78	14	0
pull-stream	769	441	271	57	280	165	95	20	774	440	278	56	781	467	248	66	763	442	266	55	149	88	54	7
q	2,035	158	1,792	85	703	46	630	27	1,856	138	1,635	83	1,958	138	1,726	94	2,007	145	1,770	92	401	38	350	13
spacl-core	239	199	39	1	80	63	17	0	211	175	35	1	187	155	31	1	214	181	32	1	25	23	2	0
zip-a-folder	100	23	3	74	39	19	17	3	98	27	3	68	97	26	4	67	101	27	3	71	20	5	1	14
Total	6,712	3,237	3,155	320	2,333	1,117	1,134	82	6,305	3,053	2,950	302	6,442	3,081	3,036	325	6,629	3,249	3,064	316	1,326	708	568	50

Table 5: Number of mutants generated using the *codellama-34b-instruct* LLM at temperature 0.0 using templates full, onemutation, noexplanation, noinstructions, gen.system prompt, basic (showing one run of each).

	full	**onemutation**	**noexplanation**	**noinstructions**	**generic sys. prmpt**	**basic**
Complex.js	4.27	3.37	5.09	4.27	4.17	11.98
countries-and-timezones	11.13	7.75	11.17	10.87	10.85	11.29
crawler-url-parser	9.50	6.41	9.46	9.49	9.30	20.04
delta	9.55	7.38	9.91	9.43	9.14	19.63
image-downloader	12.67	8.82	12.89	11.01	11.48	21.92
node-dirty	7.53	6.90	7.58	7.41	7.51	17.52
node-geo-point	8.86	6.10	8.79	7.75	8.66	15.66
node-jsonfile	9.73	6.98	9.76	7.77	8.91	11.64
plural	8.14	5.21	8.41	7.58	7.80	23.64
pull-stream	6.72	4.57	7.53	7.48	7.30	11.92
q	8.61	7.61	9.21	8.60	8.58	16.18
spacl-core	9.30	5.86	10.44	9.43	9.44	14.27
zip-a-folder	9.85	5.33	9.02	10.05	10.10	24.60

Table 6: Average string similarity of mutants to the original code fragments that they replace, for mutants generated using each of the prompt templates at temperature 0.0 using *codellama-34b-instruct*.

mutants killed and survived to be quite stable across trials (the supplemental materials provide further detail).

We also investigated how similar mutants produced using the different prompt templates are to the original code fragments they replace. As manually inspecting sufficient samples of mutants from each of the different configurations would be infeasible, we instead rely on an automated measure. We calculate the Levenshtein string edit distance for each mutant between the mutated code and the original code. Table 6 reports the average string edit distance scores for each of the prompt templates by project.

Interpreting the results across different projects is challenging, as each project uses different code idioms that might lead to different mutations. However, we observe several interesting trends by comparing the mutant similarity across prompts (within the same project). We find the *basic* template to produce the mutants that are *least similar* to the original code. We examined samples of these mutants and found that many were creative changes that injected large code blocks in place of short, simple values. For example, in *crawler-url-parser*, the mutant with the largest string edit distance (297) involves replacing a constant TRUE with an object literal. While the *onemutation* template tended to produce mutants most similar to the original code, this is likely due to the more limited sample space. We infer that prompting for multiple mutants can result in the LLM suggesting more significant code changes than it would otherwise have.

The *full* template produces the most mutants and surviving mutants overall. Using a specialized system prompt has a marginal effect. Including instructions on performing mutations and requesting explanations for mutations only modestly affects the number of mutants and surviving mutants. Requesting only one mutation dramatically reduces the number of generated and surviving mutants, and even greater reductions are observed if the LLM is only asked to fill in the placeholder without additional guidance.

4.7 RQ5: how does the effectiveness of *LLMorpheus* depend on the LLM being used?

The results discussed thus far were obtained with the *codellama-34b-instruct* LLM. To determine how the quality of results depends on the particular LLM being used; we also experimented with *codellama-13b-instruct*, *llama-3.3-70b-instruct*, *mixtral-8x7b-instruct*, and *gpt-4o-mini* at temperature 0.0.

Table 7 shows the number of mutant candidates produced using each model (along with a breakdown how many of those candidates are syntactically invalid, identical to the original code, or duplicates), and the number of mutants produced using each model, classified as killed, surviving, and timed-out. Figure 9 shows a visual comparison of the total number of mutant candidates and mutants produced using each of the five LLMs under consideration, aggregated over all 13 subject applications. From these results, it can be seen that:

- The *codellama-34b-instruct* model generates the largest number of mutant candidates (9,967), though the number of mutant candidates produced by *codellama-13b-instruct*, *mixtral-8x7b-instruct*, *llama-3.3-70b-instruct*, and *gpt-4o-mini* are quite similar. *codellama-13b-instruct* produces noticeably fewer mutant candidates (8,088).

	codellama-13b-instruct (run #354)								*mixtral-8x7b-instruct (run #360)*							
	#candidates	**#invalid**	**#identical**	**#duplicate**	**#mutants**	**#killed**	**#survived**	**#timeout**	**#candidates**	**#invalid**	**#identical**	**#duplicate**	**#mutants**	**#killed**	**#survived**	**#timeout**
Complex.js	1,410	339	116	28	955	553	401	1	1,272	310	0	15	962	589	373	0
countries-and-timezones	305	83	15	1	207	177	30	0	272	65	2	5	205	166	39	0
crawler-url-parser	494	186	51	12	247	129	118	0	411	165	0	3	234	130	104	0
delta	1,334	530	92	16	712	583	107	22	1,132	452	0	24	680	516	128	36
image-downloader	122	40	5	2	77	48	29	0	107	38	0	1	69	46	23	0
node-dirty	439	161	33	11	245	142	92	11	300	109	0	10	191	111	72	8
node-geo-point	390	64	21	16	304	237	67	0	341	88	0	11	247	166	81	0
node-jsonfile	191	43	10	7	138	43	45	50	155	23	0	4	132	54	32	46
plural	407	100	99	17	208	154	53	1	299	73	0	8	226	166	60	0
pull-stream	1,002	279	54	13	669	386	237	46	934	255	1	6	678	386	248	44
q	2,993	901	379	55	1,713	122	1,518	73	2,418	772	3	50	1,643	112	1,460	71
spacl-core	377	142	40	7	185	160	25	0	330	152	0	3	157	134	22	1
zip-a-folder	137	43	7	1	87	27	55	5	117	38	0	0	78	24	44	10
Total	9,601	2,911	922	186	5,582	2,761	2,777	209	8,088	2,540	6	140	5,402	2,600	2,686	216

	llama-3.3-70b-instruct (run #23)								*gpt-4o-mini (run #58)*							
	#candidates	**#invalid**	**#identical**	**#duplicate**	**#mutants**	**#killed**	**#survived**	**#timeout**	**#candidates**	**#invalid**	**#identical**	**#duplicate**	**#mutants**	**#killed**	**#survived**	**#timeout**
Complex.js	1,417	279	0	53	1,138	690	447	1	1,432	446	0	38	986	596	390	0
countries-and-timezones	304	64	0	14	240	207	33	0	308	99	0	9	209	171	38	0
crawler-url-parser	506	175	0	22	319	208	111	0	516	227	0	11	275	181	94	0
delta	1,333	539	0	54	794	626	130	38	1,345	636	2	40	707	564	108	35
image-downloader	122	43	0	5	79	54	25	0	123	58	0	3	65	45	20	0
node-dirty	453	121	1	10	331	168	142	21	453	188	0	9	265	154	103	8
node-geo-point	399	39	0	22	358	255	103	0	399	86	0	20	311	225	86	0
node-jsonfile	198	25	0	6	173	64	37	72	198	44	0	6	154	64	26	64
plural	427	96	0	29	331	244	87	0	428	110	5	26	313	257	55	1
pull-stream	1,044	262	0	10	782	465	265	52	1,037	302	0	16	735	420	247	68
q	3,074	855	0	80	2,219	127	2,006	86	3,084	1,287	2	69	1,795	137	1,597	61
spacl-core	383	134	0	18	236	203	32	1	392	158	0	10	215	195	20	0
zip-a-folder	145	24	0	2	121	87	5	29	143	62	0	4	81	14	7	60
Total	9,805	2,656	1	325	6,823	3,398	3,423	300	9,858	3,703	9	261	5,885	3,023	2,791	297

Table 7: Mutants generated with the *codellama-13b-instruct*, *mixtral-8x7b-instruct*, *llama-3.3-70b-instruct*, and *gpt-4o-mini* LLMs, using the following parameters: temperature: 0.0, maxTokens: 250, maxNrPrompts: 2000, template: *template-full.hb*, systemPrompt: *SystemPrompt-MutationTestingExpert.txt*, rateLimit: 0, nrAttempts: 3.

- All models produce a significant number of mutant candidates that is syntactically invalid, ranging from 3,703 in the case of *gpt-4o-mini* to 2,540 in the case of *mixtral-8x7b-instruct*.
- *codellama-13b-instruct* is the only model that produces a significant number of mutant candidates that are identical to the original code fragments that they replace (922).
- None of the models produces a significant number of mutant candidates that are duplicates.
- The number of mutants that remains after discarding the invalid, identical, and duplicate mutant candidates ranges from 5,402 in the case of *mixtral-8x7b-instruct* to 6,823 in the case of *llama-3.3-70b-instruct*, with *codellama-34b-instruct* producing almost as many valid mutants (6,712).
- *llama-3.3-70b-instruct* produces the most surviving mutants (3,423), followed by *codellama-34b-instruct* (3,155).

We also explored the variability of results produced using *codellama-13b-instruct*, *mixtral-8x7b-instruct*, *llama-3.3-70b-instruct* and *gpt-4o-mini* by conducting each experiment 5 times, and determined how many distinct mutants are produced and how many mutants occur in all five runs. We found that, at temperature 0, the results obtained with *codellama-13b-instruct* are very stable across runs, with 96.15%–100% of all mutants occurring in each of the five runs.

However, with *mixtral-8x7b-instruct*, *llama-3.3-70b-instruct* and *gpt-4o-mini* we encountered more variability. With *mixtral-8x7b-instruct*, between 34.22%–50% of mutants occur in all 5 runs, with *llama-3.3-70b-instruct*, between 28.26%–58.94% occur in all 5 runs, and with *gpt-4o-mini*, between 28.26%–58.94%. We also analyzed the variance of the number of mutants killed and survived, finding that, despite the diversity of mutants across trials, the mutation score was relatively stable. The supplemental materials include tables showing the average and standard deviation of the number of mutants killed and survived.

We also examined the string similarity of mutants produced by the five LLMs to the original code and found that the *mixtral-8x7b-instruct* model tends to generate mutants with the greatest string edit distance in the most projects. We examined the top 2 mutants with the greatest string edit distance generated by this model for each project, finding several cases of unusual completions. In *q*, mixtral's most dissimilar mutants (distance 219) replaced a string literal that referred to the function `"allResolved"` with a declaration of the same function. In *delta*, mixtral's most dissimilar mutants (distance 155) apply a `reduce` operation to an object before invoking `Object.keys` on it. We saw similar trends for mixtral across all projects, with mutants that tended to include large code declarations. Examining the mutants with the greatest string edit distance for the other four

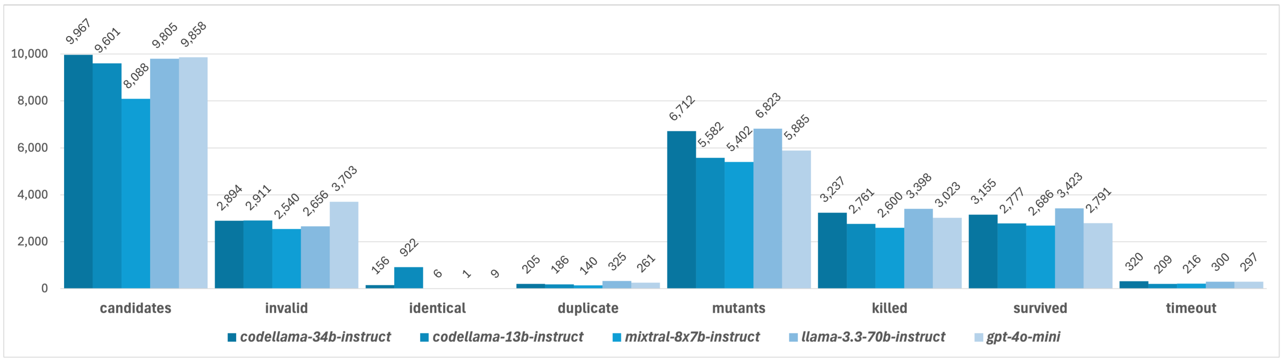


Figure 9: Comparison of the number of mutant candidates and mutants generated with the *codellama-13b-instruct*, *mixtral-8x7b-instruct*, *llama-3.3-70b-instruct*, and *gpt-4o-mini* LLMs at temperature 0.0. This chart was created from the data shown in Tables 2 and 7.

LLMs, we did not find significant trends that held across all targets. Further details can be found in the supplemental materials.

All five LLMs under consideration can be used successfully to generate large numbers of (surviving) mutants. *llama-3.3-70b-instruct* and *codellama-34b-instruct* tend to produce the largest number of surviving mutants, and *codellama-34b-instruct* tends to produce results that are stable across experiments when temperature 0 is used. *llama-3.3-70b-instruct*, *mixtral-8x7b-instruct*, and *gpt-4o-mini* produce highly variable results, even at temperature 0.

4.8 RQ6: What is the cost of running *LLMorpheus*?

The primary costs of running *LLMorpheus* are the time required to run experiments and the expenses associated with LLM usage. Regarding the latter, for the experiments reported on in this paper, we have relied on several commercial LLM service providers (octo.ai, openrouter.ai, and openai.com). Such costs tend to vary depending on the provider and the LLM being used, and are typically calculated as a function of the number of "tokens" used in the prompt and the completion[13]. The cost of commercial LLM providers also tends to vary over time, and when a newer version of an LLM is released, it often costs the same as the older version that it replaced. In our experiments, the total number of tokens used for running a full experiment with *LLMorpheus* varied by less than 20% for the five LLMs that we used[14], suggesting that token usage is a reasonable proxy for the financial costs incurred. For these reasons, we use the number of input and output tokens used in our experiments as the primary cost metric for evaluating *LLMorpheus*'s LLM usage. For completeness, we also discuss the expense in US dollars at the time of running the experiments below, but the reader should be aware that these costs are likely to vary over time.

[13]Depending on the provider, there may also be additional costs associated with the number of requests, though that was not the case for our experiments.

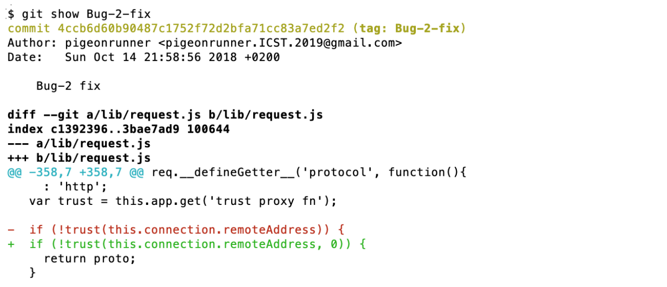
[14]Calculated from the total number of tokens reported in the Supplemental Materials associated with this paper, for experiments using the "full' prompt template at temperature 0.

project	time (sec)		#tokens		
	LLMorpheus	*StrykerJS*	prompt	compl.	total
Complex.js	3,050.00	637.85	967,508	102,517	1,070,025
countries-and-timezones	1,070.89	313.86	105,828	23,441	129,269
crawler-url-parser	1,642.70	929.43	386,223	39,175	425,398
delta	2,961.66	3,839.60	890,252	98,974	989,226
image-downloader	430.53	379.25	24,655	9,134	33,789
node-dirty	1,526.20	241.81	246,248	33,070	279,318
node-geo-point	1,411.11	987.17	316,333	30,013	346,346
node-jsonfile	690.61	474.78	57,516	14,797	72,313
plural	1,521.32	155.24	265,602	34,174	299,776
pull-stream	2,492.50	1,608.97	208,130	76,513	284,643
q	5,241.46	14,034.67	2,127,655	220,215	2,347,870
spacl-core	1,351.08	798.96	162,705	29,236	191,941
zip-a-folder	500.57	1,156.11	82,457	10,725	93,182
Total	23,890.64	25,557.70	5,841,112	721,984	6,563,096

Table 8: Results from LLMorpheus experiment (run #312). Model: *codellama-34b-instruct*, temperature: 0.0, maxTokens: 250, template: *template-full.hb*, systemPrompt: *SystemPrompt-MutationTestingExpert.txt*

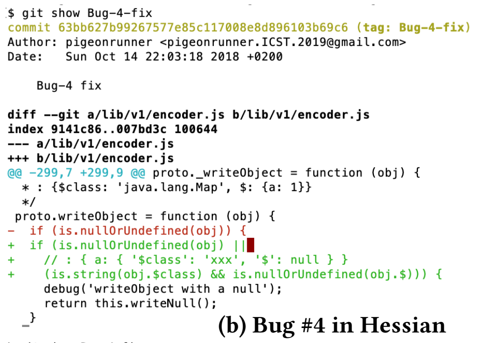
The **time** column in Table 8 shows the time needed to run *LLMorpheus* and the modified version of *StrykerJS* on each subject application. As can be seen in the table, *LLMorpheus* requires between 430.53 seconds (about 7 minutes) and 5,241.46 seconds (about 87 minutes) and *StrykerJS* between 155.24 seconds (about 2.5 minutes) and 14,034.67 seconds (about 234 minutes).

The last three columns of Table 8 show the number of tokens used in prompts and completions for each subject application and in the aggregate. From these results, it can be seen that running *LLMorpheus* required between 24,655 and 2,127,655 prompt tokens, and between 9,134 and 220,215 completion tokens. Hence, in the aggregate, 5,841,112 prompt tokens and 721,984 completion tokens were required. At the time of conducting the experiments, the cost of the *codellama-34b-instruct* LLM using octo.ai's LLM service was $0.50 per million input tokens and $1.00 per million output tokens, so for running *LLMorpheus* on all 13 applications, a total cost of approximately $3.62 was incurred. Moreover, at the time of conducting our experiments, the *llama-3.3-70b-instruct* model that we used can be accessed from $0.12 per million input tokens and $0.30 per million output tokens at openrouter.ai, and the *gpt-4o-mini* model that we used can be accessed from $0.15 per million input tokens and $0.60 per million output tokens from openai.com. Hence, a full



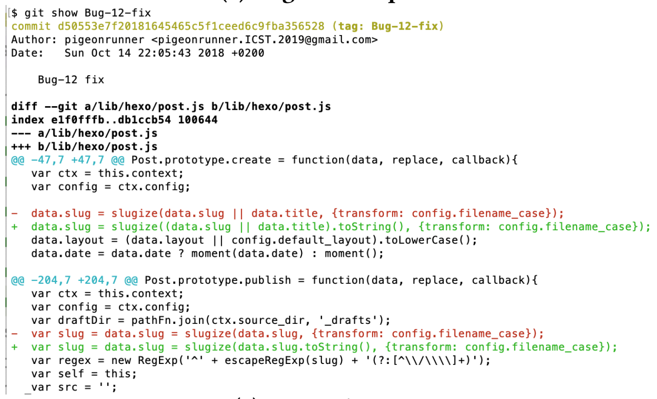
```
$ git show Bug-2-fix
commit 4ccb6d60b90487c1752f72d2bfa71cc83a7ed2f2 (tag: Bug-2-fix)
Author: pigeonrunner <pigeonrunner.ICST.2019@gmail.com>
Date:   Sun Oct 14 21:58:56 2018 +0200

    Bug-2 fix

diff --git a/lib/request.js b/lib/request.js
index c1392396..3bae7ad9 100644
--- a/lib/request.js
+++ b/lib/request.js
@@ -358,7 +358,7 @@ req.__defineGetter__('protocol', function(){
     : 'http';
   var trust = this.app.get('trust proxy fn');

-  if (!trust(this.connection.remoteAddress)) {
+  if (!trust(this.connection.remoteAddress, 0)) {
     return proto;
   }
```

(a) Bug #2 in Express



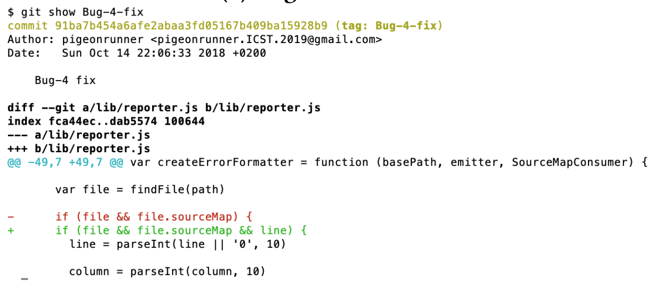
```
$ git show Bug-4-fix
commit 63bb627b99267577e85c117008e8d896103b69c6 (tag: Bug-4-fix)
Author: pigeonrunner <pigeonrunner.ICST.2019@gmail.com>
Date:   Sun Oct 14 22:03:18 2018 +0200

    Bug-4 fix

diff --git a/lib/v1/encoder.js b/lib/v1/encoder.js
index 9141c86..007bd3c 100644
--- a/lib/v1/encoder.js
+++ b/lib/v1/encoder.js
@@ -299,7 +299,9 @@ proto._writeObject = function (obj) {
  * : {$class: 'java.lang.Map', $: {a: 1}}
  */
 proto.writeObject = function (obj) {
-  if (is.nullOrUndefined(obj)) {
+  if (is.nullOrUndefined(obj) ||
+    // : { a: { '$class': 'xxx', '$': null } }
+    (is.string(obj.$class) && is.nullOrUndefined(obj.$))) {
     debug('writeObject with a null');
     return this.writeNull();
  }
```

(b) Bug #4 in Hessian



```
$ git show Bug-12-fix
commit d50553e7f20181645465c5f1ceed6c9fba356528 (tag: Bug-12-fix)
Author: pigeonrunner <pigeonrunner.ICST.2019@gmail.com>
Date:   Sun Oct 14 22:05:43 2018 +0200

    Bug-12 fix

diff --git a/lib/hexo/post.js b/lib/hexo/post.js
index e1f0fffb..db1ccb54 100644
--- a/lib/hexo/post.js
+++ b/lib/hexo/post.js
@@ -47,7 +47,7 @@ Post.prototype.create = function(data, replace, callback){
   var ctx = this.context;
   var config = ctx.config;

-  data.slug = slugize(data.slug || data.title, {transform: config.filename_case});
+  data.slug = slugize((data.slug || data.title).toString(), {transform: config.filename_case});
   data.layout = (data.layout || config.default_layout).toLowerCase();
   data.date = data.date ? moment(data.date) : moment();

@@ -204,7 +204,7 @@ Post.prototype.publish = function(data, replace, callback){
   var ctx = this.context;
   var config = ctx.config;
   var draftDir = pathFn.join(ctx.source_dir, '_drafts');
-  var slug = data.slug = slugize(data.slug, {transform: config.filename_case});
+  var slug = data.slug = slugize(data.slug.toString(), {transform: config.filename_case});
   var regex = new RegExp('^' + escapeRegExp(slug) + '(?:[^\\/\\\\]+)');
   var self = this;
   var src = '';
```

(a) Bug #2 in Hexo



```
$ git show Bug-4-fix
commit 91ba7b454a6afe2abaa3fd05167b409ba15928b9 (tag: Bug-4-fix)
Author: pigeonrunner <pigeonrunner.ICST.2019@gmail.com>
Date:   Sun Oct 14 22:06:33 2018 +0200

    Bug-4 fix

diff --git a/lib/reporter.js b/lib/reporter.js
index fca44ec..dab5574 100644
--- a/lib/reporter.js
+++ b/lib/reporter.js
@@ -49,7 +49,7 @@ var createErrorFormatter = function (basePath, emitter, SourceMapConsumer) {

       var file = findFile(path)

-      if (file && file.sourceMap) {
+      if (file && file.sourceMap && line) {
         line = parseInt(line || '0', 10)

         column = parseInt(column, 10)
```

(b) Bug #4 in Karma

Figure 10: Patches corresponding to 4 real-world bugs selected from the Bugs.js suite [25] that were used in a case study to determine *LLMorpheus*'s ability to generate mutants that resemble real-world bugs.

experiment can be run for less than $1 with *llama-3.3-70b-instruct*, and for approximately $1.30 with *gpt-4o-mini*.

It should be pointed out that the cost of the LLMs we used is significantly lower than that of larger state-of-the-art proprietary LLMs such as OpenAI's *gpt-4o*, for which https://openai.com/pricing quotes a cost of $2.50 per million input tokens and $10 per million output tokens at the time of writing. While such models might be even more capable of suggesting useful mutants, it is encouraging to see that lower-cost LLMs can achieve good results.

LLMorpheus requires between 7 and 87 minutes to generate mutants for 13 subject applications. At the time of conducting our experiments, a full experiment with *LLMorpheus* on all 13 applications costs up to $3.62 depending on the LLM being used, suggesting that cost is not a prohibitive limiting factor.

4.9 RQ7: Is *LLMorpheus* capable of producing mutants that resemble existing bugs?

To determine whether *LLMorpheus* is capable of producing mutants that resemble existing bugs, we conducted a case study involving 4 bugs from Bugs.js [25], a collection of 453 bugs from real-world JavaScript applications. For each of these bugs, the original faulty version is provided, along with a cleaned patch extracted from the bug fix and instructions on how to execute the test cases. In this study, we applied *LLMorpheus* to the *fixed* version of a program by introducing placeholders near the location of the fix, generating mutants, executing the program's tests for each of these mutants and checking if the observed test failures were identical to those caused by the original bug. Below, we report on our findings for 4 such bugs.

Express.js Bug#2. Figure 10(a) shows bug #2 in Express, a popular web framework for Node.js. This bug occurs at line 361 in the file `lib/request.js` and involves the invocation of a function `trust` with a single argument `this.connection.remoteAddress`. Here, the fix involved the addition of a second argument, `0`. Reintroducing this bug in the fixed version causes 2 tests to fail. When applied to the fixed version, *LLMorpheus* creates the following 3 mutants:

- replacing `!trust(this.connection.remoteAddress, 0)` with `trust(this.connection.remoteAddress, 1)`
- replacing `!trust(this.connection.remoteAddress, 0)` with `!trust(this.connection.remoteAddress)`
- replacing `!trust(this.connection.remoteAddress, 0)` with `trust(this.connection.localAddress, 0)`

The second mutant is identical to the original bug. The other two mutants cause multiple test failures that differ from those caused by the original bug.

We repeated the same experiment 4 more times[15], and found that in some cases *LLMorpheus* produces mutants such as `!trust(this.connection.localAddress, 1)` that differ from the original bug but cause

[15]Data for 5 experiments with each of these bugs is included with supplemental materials.

the same test failures. Moreover, in one experiment, *LLMorpheus* produced a mutant `!this.app.get('trust_proxy')` that reproduces one of the two test failures that were caused by the original bug.

Hessian.js Bug#4. Figure 10(b) shows bug #4 in Hessian, a serialization framework. This bug occurs at line 302 in file `lib/v1/encoder.js` and involves the condition of an `if`-statement. Here, the fix for the bug involves changing the condition from `is.nullOrUndefined(obj)` to `is.nullOrUndefined(obj) || (is.string(obj.$class) && is.nullOrUndefined(obj.$))`. Reintroducing this bug in the fixed version results in a test failure.

When applied to the fixed version, *LLMorpheus* creates the following 3 mutants:

- replacing `is.nullOrUndefined(obj) || (is.string(obj.$class) && is.nullOrUndefined(obj.$))` with `obj === null`,
- replacing `is.nullOrUndefined(obj) || (is.string(obj.$class) && is.nullOrUndefined(obj.$))` with `is.nullOrUndefined(obj.$)`
- replacing `is.nullOrUndefined(obj) || (is.string(obj.$class) && is.nullOrUndefined(obj.$))` with `!obj`

In this case, none of the generated mutants are identical to the original bug. However, the first and the third mutants *cause exactly the same test failures as the original bug.* The second mutant causes multiple test failures that differ from those produced by the original bug. We repeated the same experiment 4 more times and while *LLMorpheus* never reproduced the original bug, it produced mutants with the same behavior as the original bug on multiple occasions.

Hexo Bug#12. Figure 10(c) shows bug #12 in Hexo, a blogging framework for Node.js. The bug involves lines 50 and 207 in file `lib/hexo/post.js`. Here, the fix involves inserting a call to `toString` on the expression that is passed as the first argument to a function `slugize` in each of these lines. This issue causes two test failures, one corresponding to each of the lines that was changed in the fix, so it can be viewed as two similar bugs that can be fixed independently.

We decided to focus on the slightly simpler expression at line 207, where *LLMorpheus* creates the following 7 mutants:

(1) replacing `slugize((data.slug || data.title).toString(), transform: config.filename_case)` with `slugize((data.slug && data.title).toString(), transform: config.filename_case)`
(2) replacing `data.slug.toString()` with `data.slug`
(3) replacing `data.slug.toString()` with `data.title`
(4) replacing `data.slug.toString()` with `data.slug + 'test'`
(5) replacing `slugize(data.slug.toString(), transform: config.filename_case)` with `slugize(data.slug.toUpperCase())`
(6) replacing `slugize(data.slug.toString(), transform: config.filename_case)` with `slugize(data.title)`
(7) replacing `slugize(data.slug.toString(), transform: config.filename_case)` with `slugize(data.slug + Math.random())`

The second of these mutants is identical to the first part of the original bug and causes one of the two failures associated with the entire buggy code fragment. The other mutants cause multiple test failures that differ from the original bug.

4.10 Karma Bug #4

Figure 10(d) shows bug #4 in Karma, a testing framework. The bug occurs on line 46 in file `lib/reporter.js` where a buggy condition `file && file.sourceMap` in an if-statement was changed to `file && file.sourceMap && line` in the fixed version.

When applied to the fixed version, *LLMorpheus* creates the following 3 mutants at this line:

- replacing `file && file.sourceMap && line` with `file && line`,
- replacing `file && file.sourceMap && line` with `file && file.sourceMap`, and
- replacing `file && file.sourceMap && line` with `file &&!file.sourceMap && line`

The first of these mutants has the same behavior as the fixed version, i.e., it is a surviving mutant, indicating that testing may not be sufficiently rigorous. The second mutant is identical to the original bug. The third mutant causes multiple test failures that differ from those caused by the original bug.

In each of these cases, *LLMorpheus* produced either a mutant that was identical to a bug that was originally observed, or one that caused the same test failures as a bug that was originally observed. The latter case suggests that *LLMorpheus*'s ability to generate mutants that resemble real-world tests is not entirely due to training-set leakage.

In each of the experiments under consideration, *LLMorpheus* was able capable to produce mutants that resemble existing bugs.

4.11 Experimental Data

All experimental data associated with the experiments reported on in this paper can be found at https://github.com/neu-se/mutation-testing-data.

5 THREATS TO VALIDITY

The projects used to evaluate *LLMorpheus* may not be representative of the entire ecosystem of JavaScript packages. To mitigate this risk, we select popular packages used in prior JavaScript testing tool evaluations and report results per project, discussing the full range of behaviors we witness. As in many evaluations of LLM-based tools, the validity of our conclusions may be threatened by including our evaluation subjects in the training data for the models. If the model were in fact trained on bugs in some of the programs we asked it to create bugs in, one would expect its performance on those programs to vary significantly from those that it was not pretrained on. We mitigate this risk by conducting experiments with five LLMs, four of which are "open" in the sense that the training process is documented, thus enabling reproducibility and detailed analysis of experimental results.

Truly determining if a mutant is equivalent requires significant effort and despite the best efforts of two authors to rigorously evaluate them, there may be errors in our categorizing of mutants.

One of the key evaluation criteria used in previous work on mutation testing is "coupling", i.e., determining whether a test suite that detects particular mutants also detects particular real faults [22, 30, 36]. We investigated the feasibility of conducting such a study using the Bugs.js suite [25], but we found that most of these subjects could not be used at all due to their reliance on outdated versions of various libraries and because of their incompatibility with modern Node.js versions that *StrykerJS* requires, causing them to be incompatible with *LLMorpheus*. These projects also have flaky tests, making it particularly challenging to perform mutation analysis [53]. We therefore opted for conducting a case study involving 4 real-world bugs from the Bugs.js suite that we were able to reproduce reliably. The results of this case study may be skewed because the code for previous buggy versions of the applications may have been included in the training set of the LLM that we used. However, in several cases, we observed that *LLMorpheus* produced mutants that *differed from the original bug but that caused the same test failures.* This suggests that *LLMorpheus*'s ability to produce mutants that resemble real-world bugs is not entirely due to training-set leakage.

Evaluating tools that rely on LLMs face significant reproducibility challenges. We mitigate these risks by (i) evaluating *LLMorpheus* using four open LLMs that are version-controlled and permanently archived (in addition to one popular proprietary LLM), (ii) repeating each experiment 5 times and (iii) making all experimental data available as supplemental materials, and (iv) making *LLMorpheus*, our evaluation scripts and results publicly available. Including all results for all experiments in the main body of this paper would significantly decrease the readability of the work. Where we observed significant variability in results, we include data regarding that distribution in the paper directly. In all cases, the supplemental materials associated with this paper include results for all trials of all experiments and summary tables that describe the observed variability for each configuration.

Lastly, a possible concern is that *LLMorpheus* only supports JavaScript and TypeScript, and that applicability beyond these languages may be unclear. Implementing the same approach for a different language would involve various steps (parsing ASTs, executing tests, etc.) that are language-specific and would involve very significant engineering effort, but should otherwise be straightforward.

6 RELATED WORK

Mutation testing, first introduced in the 1970's [18], has a long history [46]. The era of "big code" and software repository mining has enabled the large-scale evaluation of the core hypothesis behind mutation testing, namely, that mutants are coupled to real faults. Just *et al.* mined real faults from Java applications and found a statistically significant correlation between mutation detection and real fault detection [30]. This finding has since been replicated on newer, larger datasets consisting of faults from even more Java programs [36]. Gay and Salahirad extended this methodology to examine the extent to which individual mutation operators are most coupled to real faults [22]. While this has demonstrated that test suites that detect more mutants are also likely to detect more bugs, it also underscores the need for new mutation approaches that can generate faults that are coupled to more real bugs.

ML for Mutation Testing: Several recent projects have considered the use of LLMs and other AI-based techniques for mutation testing. μBert [17, 34] resembles *LLMorpheus* in that both techniques select some designated code fragments, and query a model what they could be replaced with. μBert masks one token at a time, so its mutations involve changes to a single variable or operator. By contrast, *LLMorpheus*' placeholders correspond to (sequences of) AST nodes so it may suggest mutations involving more significant changes to complex expressions. A crucial difference between the techniques is that *LLMorpheus* utilizes prompts that provide an LLM with additional guidance whereas μBert provides no way of guiding the mutations at all, and is therefore completely at the mercy of what the model thinks masked tokens should be replaced with. In our experiments with different prompts (Section 4.6), the *basic* prompt is analogous to μBert in that it merely asks the LLM what placeholders should be replaced with. Our results show this to be much less effective at producing interesting mutants, thus demonstrating the usefulness of including additional information in prompts. Our work also differs from [17] by considering several LLMs, different temperatures and by targeting a different language.

In recent work, Garg et al. [21] explore the coupling between mutants generated using μBert and 45 reproducible vulnerabilities from the Vul4J dataset. They distinguish between *strongly coupled mutants* that fail the same tests for the same reasons as the vulnerabilities and *test coupled mutants* that fail the same tests but for some different reason as the vulnerabilities. While they find the majority (32 of 45) of μBert-generated mutants to be strongly coupled, they also find that strongly coupled mutants are scarce, representing just 1.17% of killable mutants. It would be interesting to explore whether the use of more elaborate prompting strategies such as those employed by *LLMorpheus* could be used to increase the ratio of strongly coupled mutants.

Tian et al. [56] consider the use of LLMs for determining whether mutants are equivalent and compare their effectiveness to that of traditional techniques for mutant equivalence detection. Their study considers the detection of equivalent mutants in 19 Java programs from the MutantBench suite [59], from which mutants were derived using standard mutation operators from μJava [41]. Tian et al. experimented with 10 LLMs. They consider 10 state-of-the-art LLMs and several strategies for fine-tuning and prompting, and consider three widely used traditional techniques (compiler-based, ML-based, and Tree-Based Neural NetWork) as the baseline for comparison. Their results indicate that LLMs are significantly better than traditional techniques at equivalent mutant detection, with the fine-tuned code embedding strategy being the most effective. It would be interesting to explore to what extent these results carry over to the detection of mutants that were produced using LLMs using tools such as *LLMorpheus*.

Similar to our interests, Wang et al. [60] perform an exploratory study on using large language models to generate mutants. Unlike our prompting strategy that generates up to three mutants per-AST node, Wang et al. explore a strategy that generates mutants at the granularity of entire methods. We demonstrate the nuances of prompt engineering in this context by exploring performance under different prompts (RQ4). These complementary works demonstrate the potential of using LLMs for mutation testing.

Several projects [10, 33] have considered the use of LLMs as mutation operators in the context of Genetic and Search-Based techniques to improve the efficiency of the search. Brownlee et al. [10] consider the generation of alternate implementations for methods and experimented with prompts exhibiting different levels of detail, similar to our experiments reported on in Section 4.6, finding that more detailed prompting generally improves the number of successful patches.

Several other works rely on LLMs to validate the results of mutation testing tools. Li and Shin [40] use 4 syntactic mutation operators and then observe the change to the natural language description that an LLM generates of the mutated code. MuTAP [15] uses an off-the-shelf syntactic mutation tool to generate mutants for a Python program, and then prompts an LLM to generate a test that can detect those mutants.

Equivalent Mutants: Kushigian et al. [35] study the types and prevalence of equivalent mutants in Java programs, considering why they are equivalent and how challenging it is to detect that they are. Their study considers 19 Java open-source programs from which mutants are derived using Major [29], a rule-based mutation-testing framework for Java that supports similar mutation operators as *StrykerJS*. Their findings indicate that around 3% of mutants are equivalent, and these equivalent mutants are further classified according to criteria that reflect *why* a mutant is equivalent, and *how* this could be determined. Based on these findings, Kushigian et al. propose Equivalent Mutant Suppression (EMS), a collection of simple static checks for detecting equivalent mutants.

Improving Mutation operators: Other approaches for mutation testing aim to generate mutants that represent a wider variety of faults. "Higher-order mutation" combines multiple mutations concurrently, creating more complex faults, but still limited by the set of operators implemented [23, 27]. More recently, Brown *et al.* improve mutation by mining patches for new idioms to use as mutation operators [9]. Beller *et al.* design a similar tool and evaluate it at Facebook, with the goal of increasing adoption of mutation testing [8] Taking this idea further, Tufano *et al.* create *DeepMutation*, an approach that learns models for performing mutation from real bugs [58]. This idea was refined by Tian *et al.*'s *LEAM*, which improves the search process by leveraging program grammars [55]. Patra and Pradel's *SemSeed* learns to generate mutants from fixes of real-world identifier and literal semantic bugs [47]. Unlike these approaches, *LLMorpheus* uses a *pre-trained LLM*, requiring no training to apply it to a new project.

Mutation Testing Applications and Tools: Belén Sánchez et al. [7] report on a results of a qualitative study among open-source developers on the use of mutation testing. Their findings indicate that developers find mutation testing useful for improving test suite quality, detecting bugs, and improving code maintainabiity and that performance considerations are the biggest impediment to adoption. Much of the research advancing the state of mutation testing tooling has targeted Java, such as MuJava [41], Javalanche [52], Jumble [26], Judy [42] and Major [29]. Gopinath *et al.* empirically compared two of these research-oriented tools tools (Judy [42], Major [29]) with an industry-oriented tool (Pit [14]), finding that despite the stated similarities between the tools, each produced a somewhat different set of mutants [24]. Pit is actively maintained, and the open-source tool is also available packaged with professional plugins under the name 'ArcMutate' [12]. Also aimed at practitioners, the *Stryker* mutation tool is a framework that supports code written in JavaScript, TypeScript, C#, and Scala [54]. We build *LLMorpheus* atop Stryker. Deb *et al.* examine a new, language-agnostic approach to generating mutants using regular expressions [16]. Future work may examine the feasibility of implementing *LLMorpheus* using this approach.

Mutation and Test Generation: There is a long line of research on test generation techniques that specifically target mutated code. DeMillo and Offutt [19] presented a technique that relies on solving systems of algebraic constraints to derive test cases that target mutated code. Fraser and Zeller [20] present μTEST, an approach that automatically generates unit tests for object-oriented classes based on mutation analysis. Their test generation technique uses mutations as the coverage criterion that it aims to maximize and creates tests containing oracles that test the mutated value. Chekam et al. [11] present a test generation technique based on symbolic execution that systematically searches for situations where program behaviors of the original program diverges from that of mutated versions. Lee et al. [37] present a grey-box fuzzing technique that involves executing both the original and the mutated code in the same fuzzing driver to direct the generation of test inputs towards those that kill mutants. Adapting LLM-based test generation techniques [6, 51, 57] to target mutated code would be an interested topic for future work.

LLMs and Testing: Beyond mutation testing, LLMs have also been used for test generation. Bareiß et al. [6] present an approach for test generation that follows a few-shot learning paradigm, outperforming traditional feedback-directed test generation [44]. Tufano et al. [57] present an approach for test generation using a BART transformer model [39] that is fine-tuned on a training set of functions and corresponding tests. Lemieux et al. [38] present an approach where tests generated by Codex are used to assist search-based testing techniques [45] in situations where such techniques get "stuck" because the generated test cases diverge too far from expected uses of the code under test. TestPilot [51] produces unit tests for JavaScript programs by prompting an LLM with the start of a test for an API function, with information about that function (signature, body, and usage examples mined from project documentation) embedded in code comments. In response, the LLM will produce a candidate test, which it executes to determine whether it passes or fails. In case of failure, TestPilot attempts to fix the failing test by re-prompting the LLM with the error message. In principle, *LLMorpheus* can be used to evaluate such test generation techniques by providing a means to assess the quality of the generated tests.

7 CONCLUSIONS AND FUTURE WORK

We have presented *LLMorpheus*, an LLM-based technique for mutation testing. In this approach, code fragments at designated locations in the program's source code are replaced with the word "PLACEHOLDER", and an LLM is given a prompt that includes: general background on mutation testing, the original code fragment, and instructions directing the LLM to replace the placeholder with a buggy piece of code. The mutants produced by *LLMorpheus* are passed to a modified version of the popular *StrykerJS* mutation testing tool, which runs the tests, classifies mutants, and creates an interactive web page for inspecting the results.

An empirical evaluation on 13 subject applications demonstrates that *LLMorpheus* is capable of producing mutants that resemble real bugs that cannot be produced using standard mutation operators. We found that the majority (78%) of surviving mutants produced by

LLMorpheus are behavioral changes and that 20% of them are equivalent mutants. Experiments with variations on the prompt template reveal that the "full" template that includes all information performs best and that omitting parts of the information from this template matters to varying degrees. From experiments with five LLMs, we found that *llama-3.3-70b-instruct* and *codellama-34b-instruct* generally produced the largest number of mutants and surviving mutants. Moreover, a case study involving four real-world bugs from the Bugs.js suite [25] revealed that, in each case, *LLMorpheus* produced either a mutant that was identical to a bug that was originally observed, or one that caused the same test failures as a bug that was originally observed.

The number of mutants produced by *LLMorpheus* can become quite large, and executing them can take considerable time. In future work, we plan to explore techniques for pruning and prioritizing mutants, focusing particularly on reducing the number of equivalent mutants. From a manual investigation of 105 equivalent mutants, we observed several common patterns, such as replacing an expression "!x" with "x === null" or "x === undefined" or replacing call to `String.substring` with calls to `String.substr` and `String.slice`, two methods with similar semantics. We expect that most of these equivalent mutants can be filtered out using simple AST-based analysis. However, further investigation is needed because a few of the mutants that cause behavioral differences are syntactically similar to these patterns. This means that any pattern matching based approach should consider the context in which the mutation occurs to make a determination whether a mutant is likely to be equivalent. To deal with more challenging cases, future work could also explore the use of symbolic execution or efficient formal reasoning techniques for automatically identifying mutants that are likely to be equivalent.

In our research we have used LLMs in their default configuration, without any fine-tuning. The strong results obtained with the relatively small *codellama-34b-instruct* LLM that is trained for code-related tasks suggests that fine-tuning an LLM for the specific task of mutation testing might be worthwhile, particularly with the goal of optimizing the number of mutants that are not equivalent.

REFERENCES

[1] 2025. Babel. https://babeljs.io/. Accessed 1/8/2025.

[2] 2025. countries-and-timezones. https://github.com/manuelmhtr/countries-and-timezones. Accessed 1/8/2025.

[3] 2025. Handlebars. https://handlebarsjs.com/. Accessed 1/8/2025.

[4] 2025. zip-a-folder. https://github.com/maugenst/zip-a-folder. Accessed 1/8/2025.

[5] Ellen Arteca, Max Schäfer, and Frank Tip. 2023. Learning How to Listen: Automatically Finding Bug Patterns in Event-Driven JavaScript APIs. *IEEE Trans. Software Eng.* 49, 1 (2023), 166–184.

[6] 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). https://doi.org/10.48550/ARXIV.2206.01335 arXiv:2206.01335

[7] Ana Belén Sánchez, José Antonio Parejo, Sergio Segura, Amador Durán Toro, and Mike Papadakis. 2024. Mutation Testing in Practice: Insights From Open-Source Software Developers. *IEEE Trans. Software Eng.* 50, 5 (2024), 1130–1143. https://doi.org/10.1109/TSE.2024.3377378

[8] 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.

[9] David Bingham Brown, Michael Vaughn, Ben Liblit, and Thomas Reps. 2017. The care and feeding of wild-caught mutants. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 511–522. https://doi.org/10.1145/3106237.3106280

[10] Alexander E. I. Brownlee, James Callan, Karine Even-Mendoza, Alina Geiger, Carol Hanna, Justyna Petke, Federica Sarro, and Dominik Sobania. 2023. Enhancing Genetic Improvement Mutations Using Large Language Models. In *Search-Based Software Engineering - 15th International Symposium, SSBSE 2023, San Francisco, CA, USA, December 8, 2023, Proceedings (Lecture Notes in Computer Science, Vol. 14415)*, Paolo Arcaini, Tao Yue, and Erik M. Fredericks (Eds.). Springer, 153–159.

[11] 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.

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

[13] Henry Coles. 2024. ArcMutate: Extended Mutation Operators. https://docs.arcmutate.com/docs/extended-operators.html.

[14] 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. https://doi.org/10.1145/2931037.2948707

[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.

[16] Sourav Deb, Kush Jain, Rijnard Van Tonder, Claire Le Goues, and Alex Groce. 2024. Syntax Is All You Need: A Universal-Language Approach to Mutant Generation. In *Proceedings of the ACM on Software Engineering (FSE 2024)*.

[17] Renzo Degiovanni and Mike Papadakis. 2022. μBert: Mutation Testing using Pre-Trained Language Models. In *15th IEEE International Conference on Software Testing, Verification and Validation Workshops ICST Workshops 2022, Valencia, Spain, April 4-13, 2022*. IEEE, 160–169.

[18] 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. https://doi.org/10.1109/C-M.1978.218136

[19] Richard A. DeMillo and A. Jefferson Offutt. 1991. Constraint-Based Automatic Test Data Generation. *IEEE Trans. Software Eng.* 17, 9 (1991), 900–910. https://doi.org/10.1109/32.92910

[20] Gordon Fraser and Andreas Zeller. 2010. Mutation-driven generation of unit tests and oracles. In *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010*, Paolo Tonella and Alessandro Orso (Eds.). ACM, 147–158.

[21] Aayush Garg, Renzo Degiovanni, Mike Papadakis, and Yves Le Traon. 2024. On the Coupling between Vulnerabilities and LLM-Generated Mutants: A Study on Vul4J Dataset. In *IEEE Conference on Software Testing, Verification and Validation, ICST 2024, Toronto, ON, Canada, May 27-31, 2024*. IEEE, 305–316.

[22] 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. https://doi.org/10.1109/ICST57152.2023.00021

[23] Ahmed S. Ghiduk, Moheb R. Girgis, and Marwa H. Shehata. 2017. Higher order mutation testing: A Systematic Literature Review. *Computer Science Review* 25 (2017), 29–48. https://doi.org/10.1016/j.cosrev.2017.06.001

[24] Rahul Gopinath, Iftekhar Ahmed, Mohammad Amin Alipour, Carlos Jensen, and Alex Groce. 2017. Does choice of mutation tool matter? *Software Quality Journal* 25, 3 (2017), 871–920.

[25] Péter Gyimesi, Béla Vancsics, Andrea Stocco, Davood Mazinanian, Árpád Beszédes, Rudolf Ferenc, and Ali Mesbah. 2019. BugJS: A Benchmark of JavaScript Bugs. In *Proceedings of 12th IEEE International Conference on Software Testing, Verification and Validation (ICST)*.

[26] Sean A. Irvine, Tin Pavlinic, Leonard Trigg, John G. Cleary, Stuart Inglis, and Mark Utting. 2007. Jumble Java Byte Code to Measure the Effectiveness of Unit Tests. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION '07)*. IEEE Computer Society, USA, 169–175.

[27] Yue Jia and Mark Harman. 2009. Higher Order Mutation Testing. *Inf. Softw. Technol.* 51, 10 (oct 2009), 1379–1393.

[28] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (2011), 649–678.

[29] 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. https://doi.org/10.1145/2610384.2628053

[30] 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.

[31] René Just, Bob Kurtz, and Paul Ammann. 2017. Inferring Mutant Utility from Program Context. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. Santa Barbara, CA, USA, 284–294.

[32] René Just and Franz Schweiggert. 2015. Higher accuracy and lower run time: Efficient mutation analysis using non-redundant mutation operators. *Software Testing, Verification and Reliability (JSTVR)* 25, 5-7 (Jan. 2015), 490–507.

[33] Sungmin Kang and Shin Yoo. 2023. Towards Objective-Tailored Genetic Improvement Through Large Language Models. In *IEEE/ACM International Workshop on Genetic Improvement, GI@ICSE 2023, Melbourne, Australia, May 20, 2023*. IEEE, 19–20.

[34] Ahmed Khanfir, Renzo Degiovanni, Mike Papadakis, and Yves Le Traon. 2023. Efficient Mutation Testing via Pre-Trained Language Models. *CoRR* abs/2301.03543 (2023). https://doi.org/10.48550/ARXIV.2301.03543 arXiv:2301.03543

[35] Benjamin Kushigian, Samuel J. Kaufman, Ryan Featherman, Hannah Potter, Ardi Madadi, and René Just. 2024. 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, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, Maria Christakis and Michael Pradel (Eds.). ACM, 654–665.

[36] 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.

[37] 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.

[38] 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 International Conference on Software Engineering (ICSE)*.

[39] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. 2019. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461* (2019).

[40] Ziyu Li and Donghwan Shin. 2024. Mutation-based Consistency Testing for Evaluating the Code Understanding Capability of LLMs. In *Proceedings of the IEEE/ACM 3rd International Conference on AI Engineering - Software Engineering for AI, CAIN 2024, Lisbon, Portugal, April 14-15, 2024*, Jane Cleland-Huang, Jan Bosch, Henry Muccini, and Grace A. Lewis (Eds.). ACM, 150–159.

[41] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. 2006. MuJava: a mutation system for Java. In *Proceedings of the 28th International Conference on Software Engineering* (Shanghai, China) *(ICSE '06)*. Association for Computing Machinery, New York, NY, USA, 827–830.

[42] L. Madeyski. 2010. Judy – a mutation testing tool for Java. *IET Software* 4 (February 2010), 32–42(10). Issue 1. https://digital-library.theiet.org/content/journals/10.1049/iet-sen.2008.0038

[43] Magnus Madsen, Frank Tip, and Ondrej Lhoták. 2015. Static analysis of event-driven Node.js JavaScript applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 505–519.

[44] 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.

[45] 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 Transactions on Software Engineering* 44, 2 (2018), 122–158. https://doi.org/10.1109/TSE.2017.2663435

[46] 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.

[47] Jibesh Patra and Michael Pradel. 2021. Semantic bug seeding: a learning-based approach for creating realistic bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. 906–918.

[48] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. 2021. Does Mutation Testing Improve Testing Practices?. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 910–921. https://doi.org/10.1109/ICSE43902.2021.00087

[49] 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. https://doi.org/10.1109/TSE.2021.3107634

[50] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. 2023. Please fix this mutant: How do developers resolve mutants surfaced during code review?. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 150–161. https://doi.org/10.1109/ICSE-SEIP58684.2023.00019

[51] 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.

[52] David Schuler and Andreas Zeller. 2009. Javalanche: efficient mutation testing for Java. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (Amsterdam, The Netherlands) *(ESEC/FSE '09)*. Association for Computing Machinery, New York, NY, USA, 297–298.

[53] August Shi, Jonathan Bell, and Darko Marinov. 2019. Mitigating the effects of flaky tests on mutation testing. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. 112–122.

[54] Stryker Team. 2025. Stryker Mutator. https://stryker-mutator.io. Accessed 1/8/2025.

[55] Zhao Tian, Junjie Chen, Qihao Zhu, Junjie Yang, and Lingming Zhang. 2023. Learning to Construct Better Mutation Faults. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 64, 13 pages. https://doi.org/10.1145/3551349.3556949

[56] 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, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, Maria Christakis and Michael Pradel (Eds.). 1733–1745.

[57] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2021. Unit Test Case Generation with Transformers and Focal Context. arXiv. https://www.microsoft.com/en-us/research/publication/unit-test-case-generation-with-transformers-and-focal-context/

[58] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. Learning How to Mutate Source Code from Bug-Fixes. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 301–312.

[59] Lars van Hijfte and Ana Oprescu. 2021. MutantBench: an Equivalent Mutant Problem Comparison Framework. In *14th IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2021, Porto de Galinhas, Brazil, April 12-16, 2021*. IEEE, 7–12.

[60] Bo Wang, Mingda Chen, Youfang Lin, Mike Papadakis, and Jie M. Zhang. 2024. An Exploratory Study on Using Large Language Models for Mutation Testing. arXiv:2406.09843 [cs.SE] https://arxiv.org/abs/2406.09843