An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation

Max Schäfer, Sarah Nadi, Aryaz Eghbali, Frank Tip

Abstract—Unit tests play a key role in ensuring the correctness of software. However, manually creating unit tests is a laborious task, motivating the need for automation. Large Language Models (LLMs) have recently been applied to various aspects of software development, including their suggested use for automated generation of unit tests, but while requiring additional training or few-shot learning on examples of existing tests. This paper presents a large-scale empirical evaluation on the effectiveness of LLMs for automated unit test generation without requiring additional training or manual effort. Concretely, we consider an approach where the LLM is provided with prompts that include the signature and implementation of a function under test, along with usage examples extracted from documentation. Furthermore, if a generated test fails, our approach attempts to generate a new test that fixes the problem by re-prompting the model with the failing test and error message. We implement our approach in TestPilot, an adaptive LLM-based test generation tool for JavaScript that automatically generates unit tests for the methods in a given project's API.
We evaluate TestPilot using OpenAI's *gpt3.5-turbo* LLM on 25 npm packages with a total of 1,684 API functions. The generated tests achieve a median statement coverage of 70.2% and branch coverage of 52.8%. In contrast, the state-of-the feedback-directed JavaScript test generation technique, Nessie, achieves only 51.3% statement coverage and 25.6% branch coverage. Furthermore, experiments with excluding parts of the information included in the prompts show that all components contribute towards the generation of effective test suites. We also find that 92.8% of TestPilot's generated tests have $\leq$ 50% similarity with existing tests (as measured by normalized edit distance), with none of them being exact copies. Finally, we run TestPilot with two additional LLMs, OpenAI's older *code-cushman-002* LLM and *StarCoder*, an LLM for which the training process is publicly documented. Overall, we observed similar results with the former (68.2% median statement coverage), and somewhat worse results with the latter (54.0% median statement coverage), suggesting that the effectiveness of the approach is influenced by the size and training set of the LLM, but does not fundamentally depend on the specific model.

Index Terms—test generation, JavaScript, language models

1 Introduction

Unit tests check the correctness of individual functions or other units of source code, and play a key role in modern software development [1]–[3]. However, creating unit tests by hand is labor-intensive and tedious, causing some developers to skip writing tests altogether [4].

This fact has inspired extensive research on techniques for automated test generation including fuzzing [5], [6], feedback-directed random test generation [7]–[11], dynamic symbolic execution [12]–[15], and search-based and evolutionary techniques [16], [17]. At a high level, most of these techniques use static or dynamic analysis techniques to explore control and data flow paths in the program, and then attempt to generate tests that maximize coverage. While they are often successful in generating tests that expose faults, these techniques have two major shortcomings. First, the generated tests are typically less readable and understandable than manually written tests [18], [19], especially due to the use of unintuitive variable names [20]. Second, the generated tests often lack assertions [21], or only contain very generic assertions (e.g., that a dereferenced variable must not be `null`), or too many spurious assertions [22]. While such tests can provide inspiration for manually crafting high-coverage test suites, they do not look natural and generally cannot be used verbatim.

Given these disadvantages, there has recently been increasing interest in utilizing machine learning-based code-generation techniques to produce better unit tests [23]–[29]. Specifically, these research efforts leverage LLMs that have been trained on large corpora of natural-language text and source code. We are specifically interested in *generative transformer models* that, when given a snippet of text or source code (referred to as the *prompt*), will predict text that is likely to follow it (henceforth referred to as the *completion*). It turns out that LLMs are good at producing natural-looking completions for both natural language and source code, and to some extent "understand" the semantics of natural language and code, based on the statistical relationships on the likelihood of seeing a particular word in a given context. Some LLMs such as BERT [30] or GPT-3 [31] are trained purely on text extracted from books and other public sources, while others like OpenAI Codex [32] and AlphaCode [33] are put through additional training on publicly available source code to make them better suited

- *M. Schäfer is with GitHub, UK*
  E-mail: max-schaefer@github.com
- *S. Nadi is with the University of Alberta, Canada*
  E-mail: nadi@ualberta.ca
- *A. Eghbali is with the University of Stuttgart, Germany*
  E-mail: aryaz.egh@gmail.com
- *F. Tip is with Northeastern University, USA*
  E-mail: f.tip@northeastern.edu

for software development tasks [34]–[43].

Given the properties of LLMs, it is reasonable to expect that they may be able to generate natural-looking tests. Not only are they likely to produce code that resembles what a human developer would write (including, for example, sensible variable names), but LLMs are also likely to produce tests containing assertions, simply because most tests in their training set do. Thus, by leveraging LLMs, one might hope to simultaneously address the two shortcomings of traditional test-generation techniques. On the other hand, one would perhaps not expect LLMs to produce tests that cover complex edge cases or exercise unusual function inputs, as these will be rare in the training data, making LLMs more suitable for generating regression tests than for bug finding.

There has been some exploratory work on using LLMs for test generation. For example, Bareiß *et al.* [25] evaluate the performance of Codex for test generation. They follow a few-shot learning paradigm where their prompt includes the function to be tested along with an example of another function and an accompanying test to give the model an idea of what a test should look like. In a limited evaluation on 18 Java methods, they find that this approach compares favorably to feedback-directed test generation [8]. Similarly, Tufano et al.'s ATHENATEST [26] generates tests using a BART transformer model [44] fine-tuned on a training set of functions and their corresponding tests. They evaluate on five Java projects, achieving comparable coverage to EvoSuite [17]. While these are promising early results, these approaches, as well as others [29], [45], [46], rely on a training corpus of functions and their corresponding tests, which is expensive to curate and maintain.

In this paper, we explore the feasibility of automatically generating unit tests using off-the-shelf LLMs, with no additional training and as little pre-processing as possible. Following Reynolds and McDonell [47], we posit that providing the model with input-output examples or performing additional training is not necessary and that careful prompt crafting is sufficient. Specifically, apart from test scaffolding code, our prompts contain (1) the signature of the function under test; (2) its documentation comment, if any; (3) usage examples for the function mined from documentation, if available; (4) its source code. Finally, we consider an adaptive component to our technique: each generated test is executed, and if it fails, the LLM is prompted again with a special prompt including (5) the failing test and the error message it produced, which often allows the model to fix the test and make it pass.

To conduct experiments, we have implemented these techniques in a system called TESTPILOT, an LLM-based test generator for JavaScript. We chose JavaScript as an example of a popular language for which test generation using traditional methods is challenging due to the absence of static type information and its permissive runtime semantics [11]. We evaluate our approach on 25 npm packages from various domains hosted on both GitHub and GitLab, with varying levels of popularity and amounts of available documentation. These packages have a total of 1,684 API functions that we attempt to generate tests for. We investigate the coverage achieved by the generated tests and their quality in terms of success rate, reasons for failure, and whether or not they contain assertions that actually exercise functionality from the target package (*non-trivial assertions*). We also empirically evaluate the effect of the various components of our prompt-crafting strategy as well as whether TESTPILOT is generating previously memorized tests from the LLM's training data.

Using OpenAI's current most capable and cost-effective model *gpt3.5-turbo*,[1] TESTPILOT's generated tests achieve a median statement coverage of 70.2%, and branch coverage of 52.8%. We find that a median 61.4% of the generated tests contain non-trivial assertions, and that these non-trivial tests alone achieve a median 61.6% coverage, indicating that the generated tests contain meaningful oracles that exercise functionality from the target package. Upon deeper examination, we find that the most common reason for the generated tests to fail is exceeding the two-second timeout we enforce, usually because of a failure to communicate test completion to the testing framework. We find that, on average, the adaptive approach is able to fix 15.6% of failing tests. Our empirical evaluation also shows that all five components included in the prompts are essential for generating meaningful test suites with high coverage. Excluding any of these components results in either a higher proportion of failing tests or in reduced coverage. On the other hand, while excluding usage examples from prompts reduces effectiveness of the approach, it does not render it obsolete, suggesting that the LLM is able to learn from the presence of similar test code in its training set.

Finally, from experiments conducted with the *gpt3.5-turbo* LLM, we note that high coverage is still achieved on packages whose source code is hosted on GitLab (and thus has not been part of the LLM's training data). Moreover, we find that 60.0% of the tests generated using the *gpt3.5-turbo* LLM have $\leq$ 40% similarity to existing tests and 92.8% have $\leq$ 50% similarity, with none of the tests being exact copies. This suggests that the generated tests are not copied verbatim from the LLM's training set.

In principle, the test generation approach under consideration can be used with any LLM. However, the effectiveness of the approach is likely to depend on the LLM's size and training set. To explore this factor, we further conducted experiments with two additional LLMs: the previous proprietary *code-cushman-002* [48] model developed by OpenAI and *StarCoder* [49], an LLM for which the training process is publicly documented. We observed qualitatively similar results using *code-cushman-002* (median coverage of 68.2% for statements, 51.2% for branches), and somewhat worse results using *StarCoder* (54.0% and 37.5%).

In summary, this paper makes the following contributions:

- A simple test generation technique where unit tests are generated by iteratively querying an LLM with a prompt containing signatures of API functions under test and, optionally, the bodies, documentation, and usage examples associated with such functions. The technique also features an adaptive component that includes in a prompt error messages observed when executing previously generated tests.

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

```
1  let mocha = require('mocha');
2  let assert = require('assert');
3  let pkg = require('package-under-test');
4
5  // function metadata, including signature of f
6
7  describe('test pkg', function() {
8    it('test f', function(done) {
9      // test code, terminated by done()
10   })
11 })
```

Fig. 1: Illustration of the structure of prompts and tests.

- An implementation of this technique for JavaScript in a tool called TESTPILOT, which is available as open-source software at https://github.com/githubnext/testpilot.
- An extensive empirical evaluation of TESTPILOT on 25 npm packages, demonstrating its effectiveness in generating test suites with high coverage. Our evaluation explores the following aspects:
  - Quality of the generated tests in terms of the assertions they contain, and coverage of tests that include non-trivial assertions.
  - Effect of excluding various prompt components.
  - Similarity of generated tests to existing tests.
  - Comparison against Nessie [11], a state-of-the-art feedback-directed random test generation technique for JavaScript.
  - Comparison of the effect of the underlying LLM on TESTPILOT's generated tests.

The raw data and analysis for all our experiments can be found at https://doi.org/10.6084/m9.figshare.23653371.

2 APPROACH

TESTPILOT generates tests using the popular JavaScript testing framework Mocha [50] with its BDD-style syntax in which tests are implemented as callback functions that are passed to the `it` function. Test suites consist of one or more calls to `it` that occur in a callback function that is passed to the `describe` function. Assertions are checked using the built-in Node.js `assert` module.

Figure 1 illustrates the structure of generated tests for a function f. Here, lines 1–3 are boilerplate code for importing the testing libraries and the Package under Test (PUT). These are followed by one or more commented-out lines containing function metadata included in the prompt, as we explain shortly. Lines 7–8 begin the definition of a test suite using `describe` with a single test defined as a callback function accepting a parameter `done` passed to the `it` function. The test code uses `assert` to check its assertions, and finally invokes `done()` to signal completion. This is necessary for asynchronous tests that may take multiple iterations of the JavaScript event loop to finish. Calling `done()` more than once results in a runtime error, while not calling it at all causes the test to fail with a timeout error.

The basic idea of our approach is to send the initial part of the above test skeleton up to (but not including) the start of the actual test code on Line 9 (highlighted above in blue) as a prompt to the LLM. Since LLMs are trained to complete a given code fragment, one might therefore expect it to generate the rest of the test for us. Comments can be included in the test skeleton to provide additional information about the function that may be useful to guide the LLM towards generating better tests.

2.1 TESTPILOT Architecture

Figure 2 presents the high-level architecture of TESTPILOT, which consists of five main components: Given a PUT as input, the *API explorer* identifies functions to test; the *documentation miner* extracts metadata about them; and the *prompt generator*, *test validator*, and *prompt refiner* collaborate to construct prompts for test generation, assemble complete tests from the LLM's response, run them to determine whether they pass, and construct further prompts to generate more tests. We now discuss each of these components in more detail.

API Explorer: This component analyzes the PUT to determine its API, i.e., the set of functions, methods, constants, etc. that the package exposes to clients. In JavaScript, it is very difficult to determine the API statically due to the highly dynamic nature of the language. Therefore, similar to other JavaScript test-generation work [10], [11], we pursue an approach based on dynamic analysis. In particular, we load the application's main package and apply introspection to traverse the resulting object graph and identify properties that are bound to functions. For each function, we record its access path (that is, the sequence of properties that must be traversed to reach it from the main module), its signature (which in the absence of static type information is simply a list of parameter names), and its definition (that is, its source code). The output of the API Explorer is a list of functions described by their access paths, signatures, and definitions; other API elements are ignored.

Documentation Miner: This component extracts code snippets and comments from documentation included with the PUT, and associates them with the API functions they pertain to. The aim is to collect, for each API function, comments and examples describing its purpose and intended usage. In JavaScript code bases, documentation is typically provided in the form of Markdown (`.md`) files, in which code snippets are embedded as fenced code blocks (i.e., blocks surrounded by triple backticks). We find all such blocks in all Markdown files in the code base, and associate with each function the set of all code snippets that textually contain the function's name. While this is a simple heuristic, code examples may not be complete or syntactically correct, so a more sophisticated approach relying on parsing or static analysis is not likely to work well. We also associate each API function with the doc comment (`/**. . . */`) that immediately precedes it, if any.

The remaining three components are the *prompt generator*, the *test validator*, and the *prompt refiner*, which work together to generate and validate tests for all API functions identified by the API Explorer, using the information provided by the Documentation Miner. Functions are processed one at a time, and for each function only one test is generated at a time (as opposed to generating an entire test suite at once). This is to enable us to validate each test individually without interference from other tests.

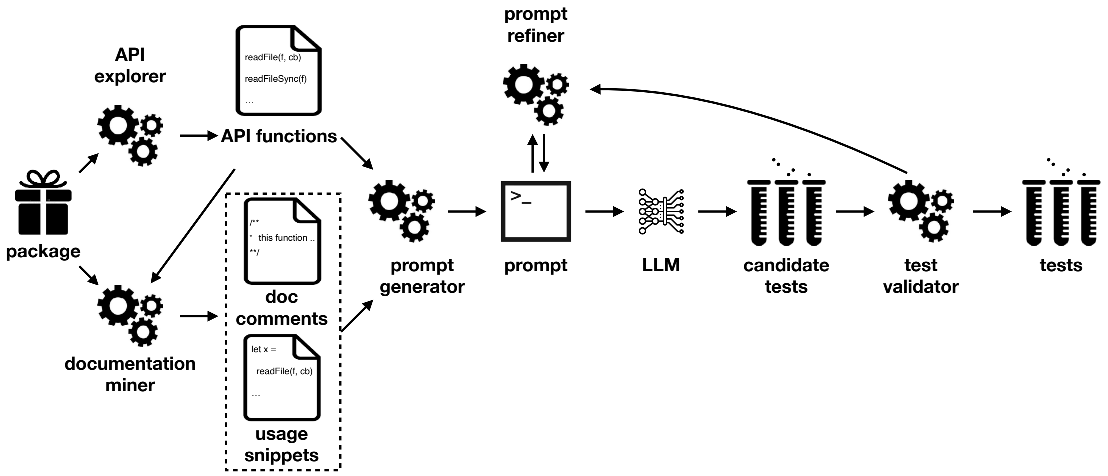

Fig. 2: Overview of the adaptive test generation technique we use in TESTPILOT.

```javascript
let mocha = require('mocha');
let assert = require('assert');
let countries_and_timezones = require('countries-and-timezones');
// countries-and-timezones.getCountry(id)
describe('test countries_and_timezones', function() {
  it('test countries-and-timezones.getCountry', function(done) {
    let country = countries_and_timezones.getCountry('US');
    assert.equal(country.name, 'United States'); // fails
    assert.equal(country.timezones.length, 2);
    assert.equal(country.timezones[0], 'America/New_York');
    assert.equal(country.timezones[1], 'America/Chicago');
    done();
  })
})
```

(a)

```javascript
let mocha = require('mocha');
let assert = require('assert');
let countries_and_timezones = require('countries-and-timezones');
// usage #1
// const ct = require('countries-and-timezones');
//
// const country = ct.getCountry('DE');
// console.log(country);
//
// /*
// Prints:
//
// {
//   id: 'DE',
//   name: 'Germany',
//   timezones: [ 'Europe/Berlin', 'Europe/Zurich' ]
// }
//
// */
// countries-and-timezones.getCountry(id)
describe('test countries_and_timezones', function() {
  it('test countries-and-timezones.getCountry', function(done) {
    let country = countries_and_timezones.getCountry('DE');
    assert.equal(country.id, 'DE');
    assert.equal(country.name, 'Germany');
    assert.equal(country.timezones[0], 'Europe/Berlin');
    assert.equal(country.timezones[1], 'Europe/Zurich');
    done();
  })
})
```

(b)

Fig. 3: Examples of prompts (highlighted) and the completions provided by the LLM, comprising complete tests. Prompt (a) contains no snippets and the test generated from it fails. Prompt (b) contains one snippet and the generated test passes.

Prompt Generator: This component constructs the initial prompt to send to the LLM for generating a test for a given function f. As mentioned above, we initially have (at most) four pieces of information about f at our disposal: its signature, its definition, its doc comment, and its usage snippets extracted from documentation. While it might seem natural to construct a prompt containing all of this information, in practice it can sometimes happen that more complex prompts lead to worse completions as the LLM gets confused by the additional information. Therefore, we follow a different strategy: we start with a very simple initial prompt that includes no metadata except the function signature, and then let the prompt refiner extend it step by step with additional information.

Test Validator: Next, we send the generated prompts to LLM and wait for completions. We only consume as many tokens as are needed to form a syntactically valid test. Since there is no guarantee that the completions suggested by the model are syntactically valid, the test validator tries to fix simple syntactic errors such as missing brackets, and then parses the resulting code to check whether it is syntactically valid. If not, the test is immediately marked as failed. Otherwise it is run using the Mocha test runner to determine whether it passes or fails (either due to an assertion error or some other runtime error).

Each returned completion can be concatenated with the prompt to yield a candidate test. However, to allow us to eliminate duplicate tests generated from different prompts, we post-process the candidate tests as follows: we strip

Algorithm 1 Pseudo-code for API exploration.

```
 1: function exploreAPI(pkgName)
 2:     modObj ← object created by importing pkgName
 3:     $seen \leftarrow \emptyset$
 4:     return explore(pkgName, modObj, seen)
 5: function explore(accessPath, obj, seen)
 6:     $apis \leftarrow \emptyset$
 7:     if $obj \notin seen$ then
 8:         $seen \leftarrow seen \cup \{ obj \}$
 9:         if obj is a function with signature sig then
10:             $apis \leftarrow apis \cup \{ \langle accessPath, sig \rangle \}$
11:         else if obj is an object then
12:             $props \leftarrow \{ prop \mid obj$ has a property $prop \}$
13:             for prop in props do
14:                 $apis \leftarrow apis \cup$
15:                     explore(extend(accessPath, prop),obj[prop], seen)
16:         else if obj is an array then
17:             for each index $i$ in the array do
18:                 $apis \leftarrow apis \cup$
19:                     explore(extend(accessPath, prop),obj[i], seen)
20:     return apis
21: function extend(accessPath, component)
22:     if component is numeric then
23:         return accessPath[component ]
24:     else
25:         return accessPath.component
```

out the comment containing the function metadata in the prompt and replace the descriptions in the `describe` and `it` calls with the generic strings `'test suite'` and `'test case'`, respectively.

Prompt Refiner: The Prompt Refiner applies a number of strategies to generate additional prompts to use for querying the model. Overall, we employ four prompt refiners as follows:

1) *FnBodyIncluder*: If p did not contain the definition of f, a prompt is created that includes it.
2) *DocCommentIncluder*: If f has a doc comment but p did not include it, a prompt with the doc comment is created.
3) *SnippetIncluder*: If usage snippets for f are available but p did not include them, a prompt with snippets is created.
4) *RetryWithError*: If t failed with error message e, a prompt is constructed that consists of: the text of the failing test t followed by a comment `// the test above fails with the following error:` e, followed by a comment `// fixed test`. This strategy is only applied once per prompt, so it is not attempted if p itself was already generated by this strategy.

The refined prompt is then used to construct a test in the same way as the original prompt. All strategies are applied independently and in all possible combinations, but note that the first three will only apply at most once and the fourth will never apply twice in a row, thus ensuring termination.

2.2 Algorithm Details

We now provide additional detail on the two key steps of our approach: API exploration and test generation.

API Exploration: Algorithm 1 shows pseudocode that illustrates how the set of functions that constitute the API for a package is identified. The algorithm takes a package under test, *pkgName*, and produces a list of pairs $\langle a, sig \rangle$ representing its API. Here, a is an *access path* that uniquely represents an API method, and *sig* is the signature of a function. Our notion of an access path takes a somewhat simplified form compared to the original concept proposed by Mezzetti et al. [51], and consists of a package name followed by a sequence of property names.

We rely on a dynamic approach to explore the API of a package *pkgName*, by creating a small program that imports the package (line 2), and relying on JavaScript's introspective capabilities to determine which properties are present in the *package root* object *modObj* that is created by importing *pkgName* and what types these properties have. Exploration of *modObj*'s properties is handled by a recursive function *explore* that begins at the access path representing the package root and that traverses this object recursively, calling another auxiliary function *extend* to extend the access path as the traversal descends into the object's structure.

During exploration, if an object is encountered at access path a whose type is a function with signature *sig*, then a pair $\langle a, sig \rangle$ is recorded (line 10). If the type of p is an object, then the objects referenced by its properties are recursively explored (lines 15–15), and if the type of p is an array, then p's properties are explored recursively as well (lines 17–19).

Test Generation: Algorithm 2 shows pseudo-code for the test generation step. The algorithm begins by initializing the set *prompts* of generated prompts, the set *tests* of generated passing tests, and the set *seen* containing all generated tests to the empty set and by using Algorithm 1 to obtain the set *apis* of (access path, signature) pairs that constitute the package's API (lines 2–5). Then, on lines 6–7, for each such pair, a base prompt is constructed and added to *prompts*, containing only the access path and signature, using the template illustrated in Figure 1. Next, lines 9–27 create additional prompts by adding the function body, example usage snippets, and documentation comments extracted from the code to previously generated prompts. Here, the *refine* function extends a previously generated prompt by adding the function body, example snippets, or doc comment. The order in which each type of information, if included, appears in prompts is fixed as follows: example snippets, error message from previously generated test, doc comments, function body, signature.

The while loop on lines 29–44 describes an iterative process for generating tests that continues as long as prompts remain that have not been processed. In each iteration, a prompt is selected and removed from *prompts*, and the LLM is queried for completions (line 31). For each *completion* that was received, a test is constructed by concatenating the prompt and the completion (line 33) and minor syntactic problems are fixed such as adding missing '}' characters at the end of the test (line 34). Moreover, we remove comments from the test to enable deduplication of tests that only differ in their comments (line 35).

If the resulting test is syntactically valid and the same test was not encountered previously, it is executed (line 38). Otherwise, we do not re-execute it but still link the prompt to the previously seen test. If the test executed successfully, we add it to *tests* (line 40). If it failed (due to an assertion failure, nontermination, or because of an uncaught exception), and if the test was not derived from a prompt that was constructed from a previous failing test (line 42), then we create a new prompt containing the failed test and the

Algorithm 2 Pseudo-code for test generation.

```
1:  function generateTests(pkgName, LLM)
2:      $prompts \leftarrow \emptyset$
3:      $tests \leftarrow \emptyset$
4:      $seen \leftarrow \emptyset$
5:      $apis \leftarrow exploreAPI(pkgName)$                                   ▷ See Algorithm 1
6:      for $api \in apis$ do
7:          $prompts \leftarrow prompts \cup \{\ createBasePrompt(api)\ \}$     ▷ create base prompts containing only the signature, see Figure 1
8:
9:      $promptsWithFnBody \leftarrow \emptyset$                                ▷ refine prompts by adding function body
10:     for $prompt \in prompts$ do
11:         $body \in findFnBody(prompt.api.accessPath, prompt.api.sig)$
12:         $promptsWithFnBody \leftarrow promptsWithFnBody \cup refine(prompt, body)$
13:     $prompts \leftarrow prompts \cup promptsWithFnBody$
14:
15:     $promptsWithExamples \leftarrow \emptyset$                              ▷ refine prompts in cases where example snippets are available
16:     for $prompt \in prompts$ do
17:         $snippets \leftarrow findExamples(prompt.api.accessPath, prompt.api.sig)$
18:         if $snippets \neq \emptyset$ then
19:             $promptsWithExamples \leftarrow promptsWithExamples \cup refine(prompt, snippets)$
20:     $prompts \leftarrow prompts \cup promptsWithExamples$
21:
22:     $promptsWithDocComments \leftarrow \emptyset$                           ▷ refine prompts in cases where doc comments are available
23:     for $prompt \in prompts$ do
24:         $docComment \leftarrow findDocComments(prompt.api.accessPath, prompt.api.sig)$
25:         if $docComment \neq \emptyset$ then
26:             $promptsWithDocComments \leftarrow promptsWithDocComments \cup refine(prompt, docComment)$
27:     $prompts \leftarrow prompts \cup promptsWithDocComments$
28:
29:     while $prompts \neq \emptyset$ do
30:         select and remove $prompt$ from $prompts$
31:         $completions \leftarrow getCompletions(LLM, prompt.text)$          ▷ request completions from the LLM
32:         for $completion \in completions$ do
33:             $test \leftarrow concatenate$(prompt, completion)
34:             $test \leftarrow fixMinorSyntaxIssues$(test)                   ▷ e.g., add missing close parentheses
35:             $test \leftarrow removeComments$(test)
36:             if $test$ is syntactically valid and $test \notin seen$ then
37:                 $seen \leftarrow seen \cup \{\ test\ \}$
38:                 $result \leftarrow executeTest(test)$                      ▷ execute the test
39:                 if $result.status = ok$ then                               ▷ add successful test to tests
40:                     $tests \leftarrow tests \cup \{\ test\ \}$
41:                 else     ▷ $result.status = assertionFailure$ or $result.status = crash$ or $result.status = nonTermination$
42:                     if $prompt$ was not constructed from a previous failed test then    ▷ apply error retry refiner
43:                         $prompt' \leftarrow refineFromError(test, result.errorMessage)$
44:                         $prompts \leftarrow prompts \cup \{\ prompt'\ \}$
45:     return $tests$
```

error message and add it to *prompts*.

When the iterative process concludes, the set *tests* is returned (line 45).

2.3 Examples

To make the discussion more concrete, we will now show two examples of how TESTPILOT generates tests.

As the first example, we consider the npm package `countries-and-timezones`.[2] API exploration reveals that this package exports a function `getCountry` with a single parameter `id` and the project's `README.md` file provides a usage example.

Figure 3(a) shows a test for this function generated from the initial highlighted prompt that only includes the function signature, but no other metadata. This test fails when execution reaches the assertion on line 8 because the expression `country.name` evaluates to `"United States of America"`, which differs from the value `"United States"` expected by the assertion.

Next, we refine this prompt to include the usage snippet as shown in the highlighted part of Figure 3(b). This enables the LLM to generate a test incorporating the information provided in this snippet, which passes when executed.

We show another example in Figure 4 from `quill-delta`,[3] a package for representing and manipulating changes to documents. As before, Figure 4(a) shows the initial prompt for `quill-delta`'s `concat` method, which concatenates two change sets, and a test that was generated from this prompt. It is noteworthy that the LLM was able to generate a syntactically correct test for `quill-delta`, where arguments such as

```
[{ insert: 'Hello' },
 { insert: ' ', attributes: { bold: true } },
 { insert: 'World!' }]
```

are passed to the constructor *even in the absence of any usage examples*. Most likely, this is because `quill-delta` is a popular package with more than 1.2 million weekly downloads, which means that the LLM is likely to have seen examples of its use in its training set.

[2] See https://www.npmjs.com/package/countries-and-timezones.

[3] See https://github.com/quilljs/delta.

```
1  let mocha = require('mocha');
2  let assert = require('assert');
3  let quill_delta = require('quill-delta');
4  // quill-delta.prototype.concat(other)
5  describe('test quill_delta', function() {
6    it('test quill-delta.prototype.concat', function(done) {
7      let delta1 = new quill_delta([{ insert: 'Hello' },
8                                    { insert: ' ',
9                                      attributes: { bold: true } },
10                                   { insert: 'World!' }]);
11     let delta2 = new quill_delta([{ insert: 'Hello' },
12                                   { insert: ' ',
13                                     attributes: { bold: true } },
14                                   { insert: 'World!' }]);
15     let delta3 = delta1.concat(delta2);
16     assert.equal(delta3.ops.length, 6); // fails
17     done();
18   })
19 })
```

(a)

```
let mocha = require('mocha');
let assert = require('assert');
let quill_delta = require('quill-delta');
// quill-delta.prototype.concat(other)
describe('test quill_delta', function() {
  it('test quill-delta.prototype.concat', function(done) {
    let delta1 = new quill_delta([{ insert: 'Hello' },
                                  { insert: ' ',
                                    attributes: { bold: true } },
                                  { insert: 'World!' }]);
    let delta2 = new quill_delta([{ insert: 'Hello' },
                                  { insert: ' ',
                                    attributes: { bold: true } },
                                  { insert: 'World!' }]);
    let delta3 = delta1.concat(delta2);
    assert.equal(delta3.ops.length, 6);
    done();
  })

  // the test above fails with the following error:
  //   expected 5 to equal 6
  // fixed test:
  it('test quill_delta', function(done) {
    let delta1 = new quill_delta([{ insert: 'Hello' },
                                  { insert: ' ',
                                    attributes: { bold: true } },
                                  { insert: 'World!' }]);
    let delta2 = new quill_delta([{ insert: 'Hello' },
                                  { insert: ' ',
                                    attributes: { bold: true } },
                                  { insert: 'World!' }]);
    let delta3 = delta1.concat(delta2);
    assert.equal(delta3.ops.length, 5);
    done();
  })
}
```

(b)

Fig. 4: Example illustrating how a prompt is refined in response to the failure of a previously generated test.

Nevertheless, the test in Figure 4(a) fails because when reaching the assertion on line 16 `delta3.ops.length` has the value `5`, whereas the assertion expects the value `6`. The reason for the assertion's failure is the fact that the `concat` method merges adjacent elements if they have the same attributes. Therefore, when execution reaches line 16, the array `delta3.ops` will hold the following value:

```
[
  { insert: 'Hello' },
  { insert: ' ', attributes: { bold: true } },
  { insert: 'World!Hello' },
  { insert: ' ', attributes: { bold: true } },
  { insert: 'World!' }
]
```

and therefore `delta3.ops.length` will have the value `5`.

In response to this failure, the Prompt Refiner will create the prompt shown in Figure 4(b) from which a passing test is generated. In this test, the expected value in the assertion has been updated to `5`, as per the assertion error message.

Note that all these tests look quite natural and similar to tests that a human developer might write, and they exercise typical usage scenarios (rather than edge cases) of the functions under test.

3 Research Questions & Evaluation Setup

3.1 Research Questions

Our evaluation aims to answer the following research questions.

RQ1 *How much statement coverage and branch coverage do tests generated by* TestPilot *achieve?* Ideally, the generated tests would achieve high coverage to ensure that most of the API's functionality is exercised. Given that our goal is to generate complete unit test suites (as opposed to bug finding), we measure statement coverage for passing tests *only*. We report coverage on both the package level and function level.

RQ2 *How does* TestPilot*'s coverage compare to Nessie [11]?* We compare TestPilot's coverage to the state-of-the-art JavaScript test generator, Nessie, which uses a feedback-directed approach.

RQ3 *How many of* TestPilot*'s generated tests contain non-trivial assertions?* A test with no assertions or with *trivial* assertions such as `assert.equal(true, true)` may still achieve high coverage. However, such tests do not provide useful oracles. We examine the generated tests and measure the prevalence of non-trivial assertions.

RQ4 *What are the characteristics of* TestPilot*'s failing tests?* We investigate the reasons behind any failing generated test.

RQ5 *How does each of the different types of information included in prompts contribute to the effectiveness of* TestPilot*'s generated tests?* To investigate if all the information included in prompts through the refiners is necessary to generate effective tests, we disable each refiner and report how it affects the results.

RQ6 *Are* TestPilot*'s generated tests copied from existing tests?* Since *gpt3.5-turbo* is trained on GitHub code, it

TABLE 1: Overview of npm packages used for evaluation, ordered by descending popularity in terms of downloads/wk. The top 10 packages correspond to the Nessie benchmark, the next 10 are additional GitHub-hosted packages we include, while the last 5 are GitLab-hosted packages.

Package	Domain	LOC	Existing Tests	Weekly Downloads	API functions			Total Examples
					#	# (%) w/ examples	# (%) w/ comment	
glob	file system	314	22	103M	21	2 (9.5%)	0 (0.0%)	4
fs-extra	file system	822	417	79M	172	23 (13.4%)	0 (0.0%)	27
graceful-fs	file system	208	11	48M	137	1 (0.7%)	0 (0.0%)	1
jsonfile	file system	46	43	48M	4	4 (100.0%)	0 (0.0%)	14
bluebird	promises	3.1K	238	26M	115	59 (51.3%)	0 (0.0%)	248
q	promises	736	214	14M	98	29 (29.6%)	15 (15.3%)	64
rsvp	promises	565	171	8.6M	29	11 (37.9%)	16 (55.2%)	15
memfs	file system	2.2K	265	13M	376	21 (5.6%)	7 (1.9%)	26
node-dir	file system	244	55	6M	6	6 (100.0%)	5 (83.3%)	8
zip-a-folder	file system	25	5	95K	3	2 (66.7%)	0 (0.0%)	2
js-sdsl	data structures	1.5K	88	9.7M	133	3 (2.3%)	0 (0.0%)	1
quill-delta	document changes	395	180	1.6M	36	17 (47.2%)	0 (0.0%)	17
complex.js	numbers/arithmetic	393	21	497K	52	7 (13.5%)	52 (100.0%)	5
pull-stream	streams	308	31	78K	24	7 (29.2%)	0 (0.0%)	7
countries-and-timezones	date & timezones	78	31	115K	7	7 (100.0%)	0 (0.0%)	7
simple-statistics	statistics	917	307	103K	89	3 (3.4%)	88 (98.9%)	3
plural	text processing	53	14	18K	4	3 (75.0%)	0 (0.0%)	3
dirty	key-value store	89	24	9.7K	27	5 (18.5%)	0 (0.0%)	2
geo-point	geographical coordinates	76	10	1.1K	19	10 (52.6%)	0 (0.0%)	11
uneval	serialization	31	3	417	1	1 (100.0%)	0 (0.0%)	1
image-downloader	image handling	32	12	23K	1	1 (100.0%)	0 (0.0%)	3
crawler-url-parser	URL parser	100	185	5	3	3 (100.0%)	0 (0.0%)	4
gitlab-js	API wrapper	205	14	184	37	4 (10.8%)	2 (5.4%)	7
core	access control	136	16	1	20	6 (30.0%)	0 (0.0%)	2
omnitool	utility library	1.6K	420	1	270	15 (5.6%)	80 (29.6%)	9

is likely that the LLM has already seen the tests of our evaluation packages before and may simply be producing copies of tests it "memorized". We investigate the similarity between the generated tests and any existing tests in our evaluation packages.

RQ7 *How much does the coverage of* TestPilot*'s generated tests rely on the underlying LLM?* To understand the generalizability of an LLM-based test generation approach and the effect of the underlying LLM TestPilot relies on, we compare coverage we obtain using *gpt3.5-turbo* with two other LLMs: (1) OpenAI's *code-cushman-002* model [48], one of *gpt3.5-turbo*'s predecessors which is part of the Codex suite of LLMs [52] and which served as the main model behind the first release of GitHub Copilot [38], and (2) *StarCoder* [49], a publicly available LLM for which the training process is fully documented.

3.2 Evaluation Setup

To answer the above research questions, we use a benchmark of 25 npm packages. Table 1 shows the size and number of downloads (popularity) of each of these packages. The first 10 packages shown in the table are the same GitHub-hosted packages used for evaluating Nessie [11], a recent feedback-directed test-generation technique for JavaScript. However, we notice that these 10 packages primarily focus on popular I/O-related libraries with a callback-heavy style, so we add 10 new packages from different domains (e.g., document processing and data structures), programming styles (primarily object-oriented), as well as less popular packages. Since *gpt3.5-turbo* (as well as the other LLMs we experiment with in RQ7) was trained on GitHub repositories, we have to assume that all our subject packages (and in particular their tests) were part of the model's training set. For this reason, we also include an additional 5 packages whose source code is hosted on GitLab.[4]

Table 1 shows that the 25 packages vary in terms of popularity (downloads/week) and size (LOC), as well as in terms of the number of API functions they offer and the extent of the available documentation. The "API functions" columns show the number of available API functions; the number and proportion of API functions that have at least one example code snippet in the documentation ("w/ examples"); and the number and proportion of API functions that have a documentation comment ("w/ comment"). We also show the total number of example snippets available in the documentation of each package.

To answer RQ1–RQ6, we run TestPilot using the *gpt3.5-turbo* LLM (version *gpt-3.5-turbo-0301*), sampling five completions of up to 100 tokens at temperature zero,[5] with all other options at their default values. In RQ7, we use the same settings for *code-cushman-002* and *StarCoder*, except that the sampling temperature for the latter is 0.01 since it does not support a temperature of zero.

Note that LLM-based test generation does not have a test-generation budget per se since it is not an infinite process. Instead, we ask the LLM for at most five completions for every prompt (but the model may return less). We deduplicate the returned tests to avoid inflating the number of generated tests. For example, if two prompts return the same test (modulo comments), we only record this test once but keep track of which prompt(s) resulted in its generation.

While we set the sampling temperature as low as possible, there is still some nondeterminism in the received responses. Accordingly, we run all experiments 10 times. All the per-package data points reported in Section 4 are median values over these 10 runs, except for integer-typed data such as number of tests where we use the ceiling of the median value. For RQ6, without loss of generality, we present the similarity numbers based on the first run only.

[4] We checked similarly-named repos to ensure that they are not mirrored on GitHub.

[5] Intuitively speaking, the sampling temperature controls the randomness of the generated completions, with lower temperatures meaning less non-determinism. Language models encode their input and output using a vocabulary of tokens, with commonly occurring sequences of characters (such as `require`, but also contiguous runs of space characters) represented by a single token.

TABLE 2: Statement and branch coverage for TESTPILOT's passing tests, generated using *gpt3.5-turbo*. We also show passing tests that uniquely cover a statement. The last two columns show Nessie's statement and branch coverage for each package. Note that Nessie generates 1000 tests per package and the reported coverage is for all generated tests.

Project	**Loading Coverage**		**TESTPILOT**					**Nessie 1000 Tests**	
	Stmt Cov	**Branch Cov**	**Total Tests**	**Passing Tests (%)**	**Stmt Cov**	**Branch Cov**	**Uniquely Contr. (%)**	**Stmt Cov**	**Branch Cov**
glob	7.0%	0.4%	68	18 (26.5%)	**71.3%**	**66.3%**	4 (22.2%)	39.7%	14.8%
fs-extra	16.8%	0.9%	471	277 (58.8%)	**58.8%**	**38.9%**	17 (6.1%)	38.0%	24.9%
graceful-fs	28.6%	9.8%	345	177 (51.4%)	49.3%	33.3%	1 (0.6%)	**49.8%**	**34.9%**
jsonfile	19.1%	0.0%	13	6 (48.0%)	38.3%	29.4%	0 (0.0%)	**91.5%**	**81.0%**
bluebird	23.7%	7.8%	370	204 (55.2%)	**68.0%**	**50.0%**	26 (12.5%)	43.8%	24.6%
q	22.4%	9.1%	323	186 (57.6%)	**70.4%**	53.7%	20 (10.5%)	66.8%	**54.4%**
rsvp	16.4%	12.6%	109	70 (64.2%)	**70.1%**	**55.3%**	6 (7.9%)	52.8%	47.0%
memfs	29.3%	7.2%	1037	471 (45.4%)	**81.1%**	**58.9%**	40 (8.5%)	64.6%	36.2%
node-dir	5.9%	0.0%	40	19 (48.1%)	64.3%	50.8%	4 (21.1%)	**65.4%**	**54.3%**
zip-a-folder	16.0%	0.0%	11	6 (54.5%)	84.0%	50.0%	0 (0.0%)	**88.0%**	**100.0%**
js-sdsl	7.9%	3.7%	409	46 (11.3%)	**33.9%**	**24.3%**	18 (39.1%)	8.5%	4.8%
quill-delta	8.1%	1.6%	152	33 (21.7%)	**73.0%**	**64.3%**	8 (24.2%)	9.6%	2.5%
complex.js	8.4%	4.6%	209	121 (58.0%)	**70.2%**	**46.5%**	10 (8.3%)	8.6%	5.4%
pull-stream	18.1%	0.0%	83	34 (41.0%)	**69.1%**	**52.8%**	11 (32.4%)	38.5%	23.8%
countries-and-timezones	4.9%	0.0%	28	13 (46.4%)	93.1%	69.1%	2 (15.4%)	**96.0%**	**80.8%**
simple-statistics	2.6%	0.0%	353	250 (70.9%)	**87.8%**	**71.3%**	14 (5.4%)	57.8%	66.0%
plural	53.8%	0.0%	13	8 (61.5%)	**73.8%**	**59.1%**	1 (12.5%)	59.2%	9.1%
dirty	4.7%	0.0%	70	32 (45.3%)	**74.5%**	**65.4%**	2 (6.3%)	4.7%	0.0%
geo-point	12.2%	0.0%	76	50 (65.8%)	**87.8%**	**70.6%**	1 (2.0%)	13.3%	0.0%
uneval	9.4%	0.0%	7	2 (28.6%)	68.8%	58.3%	0 (0.0%)	–	–
image-downloader	24.2%	0.0%	5	4 (80.0%)	**63.6%**	**50.0%**	0 (0.0%)	30.3%	22.2%
crawler-url-parser	7.2%	1.3%	14	2 (14.3%)	51.4%	35.0%	2 (100.0%)	**73.9%**	**64.1%**
gitlab-js	26.9%	0.6%	141	14 (9.9%)	51.7%	16.5%	7 (46.4%)	**55.3%**	**26.4%**
core	16.1%	0.0%	85	13 (15.3%)	**78.3%**	**50.0%**	5 (38.5%)	18.9%	0.0%
omnitool	19.2%	0.6%	1033	330 (31.9%)	**74.2%**	**55.2%**	90 (27.2%)	56.0%	28.3%
Median	16.1%	0.4%		48.0%	**70.2%**	**52.8%**	10.5%	51.3%	25.6%

We use Istanbul/nyc [53] to measure statement and branch coverage and use Mocha's default time limit of 2s per test.

4 EVALUATION RESULTS

4.1 RQ1: TESTPILOT's Coverage

Table 2 shows the number of tests TESTPILOT generates for each package, the number (and proportion) of passing tests, and the corresponding coverage achieved by the passing tests. The first two columns of Table 2 also show the coverage obtained by simply loading the package (*loading coverage*). This is the coverage we get "for free" without having any test suite, which we provide as a point of reference for interpreting our results. Overall, 9.9%–80.0% of the tests generated by TESTPILOT are passing tests, with a median of 48.0% across all packages. We now discuss the different coverage measurements of these passing tests.

Statement Coverage: The statement coverage per package achieved by the passing tests ranges between 33.9% and 93.1%, with a median of 70.2%. We note that across all packages, the achieved statement coverage is much higher than the loading coverage with a difference of 19.1%– 88.2% and a median difference of 53.7%.[6]

The lowest statement coverage TESTPILOT achieves is on `js-sdsl`, at 33.9%. Upon further investigation of this package, we find that it maintains the documentation examples that appear on its website as markdown files in a separate repository.[7] Including the extracted example snippets from this external repository increases the achieved coverage to 43.6%, which suggests the importance of including usage examples in the prompts. We examine the effect of the information included in prompts in detail in **RQ5** (Section 4.5).

It is worth noting that TESTPILOT's coverage for the GitLab projects listed in the bottom 5 rows of Table 2 ranges from 51.4% to 78.3%. This demonstrates that TESTPILOT is effective at generating high-coverage unit tests for packages it has not seen in its training set.

Branch Coverage: We also show the branch coverage achieved by the passing tests in Table 2. We find that the branch coverage per package is between 16.5% and 71.3%, with a median of 52.8%. Similar to statement coverage, the achieved branch coverage is also much higher than the loading coverage with a difference of 15.9%– 71.3% and a median difference of 50.0%.

Since achieving branch coverage is generally harder than achieving statement coverage, it is expected that the branch coverage for the generated tests is lower than the statement coverage. However, we note an interesting case in `gitlab-js` where this difference seems more pronounced (51.7% vs. 16.5%). Upon further investigation of its source code and documentation, we find that `gitlab-js` offers various configuration options and parameters to specify the GitLab repository to connect to and use/query (e.g., its url, authentication token, search parameters to use for a query). The processing of these options is reflected in the main branching logic in the code. While TESTPILOT does attempt to generate reasonable tests that call different endpoints with different options, it sometimes struggles to find the correct function call to use, resulting in type errors. In general, a large proportion of the tests TESTPILOT generates for this package fail, and thus do not contribute to our coverage numbers. It is also worth noting that properly testing such a package would require mocking, but we did not observe any of the generated tests to use mocking. In the future, it would be interesting to investigate if including mocking libraries in the prompt, or other mocking related information, may result in the model using mocking when needed.

[6] For some of the projects we share with Nessie, our loading coverage differs from the one reported in their paper. We contacted the authors, who confirmed that with recent versions of Istanbul/nyc they obtained the same numbers as we did, except for a very small difference on `memfs` (29.1% vs 29.3%), which may be due to platform differences.

[7] https://github.com/js-sdsl/js-sdsl.github.io/tree/main/start

Coverage per function: Figure 5 shows the distribution of statement coverage per function for each package. Each box corresponds to one of our benchmark packages and each data point in a box represents the statement coverage for a function in that package. The median statement coverage per function for each package is shown in red.

Overall, the median statement coverage per function for a given project ranges from 0.0%–100.0%, with a median of 77.1%. To ensure that TESTPILOT is not generating high coverage tests only for smaller functions, we run a Pearson's correlation test between the statement coverage per function and the corresponding function size (in statements). We find no statistically significant correlation between coverage and size, indicating that TESTPILOT is not only doing well for smaller functions.[8]

As expected, Figure 5 shows that for most packages, TESTPILOT does well for some functions while achieving low coverage for others. Let us take `jsonfile` as an example. In Table 2, we saw that its statement coverage at the package level is 38.3%. From Figure 5, we see that statement coverage per function ranges from 0% to 100%, with a median of almost 50%. Diving into the data, we find that there are two functions that TESTPILOT cannot cover, because their corresponding generated tests fail either due to references to non-existent files TESTPILOT includes in the tests or because they time out. However, the functions that TESTPILOT is able to cover have statement coverage ranging from 58%-100%. We can observe similar behavior with other file system dependent packages, such as `graceful-fs` or `fs-extra`. At the other end of the spectrum, we see `zip-a-folder` where TESTPILOT achieves both high statement coverage at the package level (84%) as well as high statement coverage at the function level in Figure 5 where the minimum per function coverage is 75%.

Uniquely Contributing Tests: To further understand the diversity of the generated tests, Table 2 also shows how many of the tests TESTPILOT generates are *uniquely contributing*, meaning that they cover at least one statement that no other tests cover. A median of 10.5% of the passing tests are of this kind, with some packages as high as 100.0%. These results are promising because they show that TESTPILOT can generate tests that cover edge cases, but there is clearly some redundancy among the generated tests. Of course, we cannot simply exclude all 89.5% remaining tests without losing coverage, since some statements may be covered by multiple tests non-uniquely. Exploring test suite minimization techniques [54] to reduce the size of the generated test suite is an interesting avenue for future work.

4.2 RQ2 TESTPILOT vs. Nessie

We compare TESTPILOT's coverage to the state-of-the-art JavaScript test generator Nessie [11], which uses a traditional feedback-directed approach.[9] For each package,

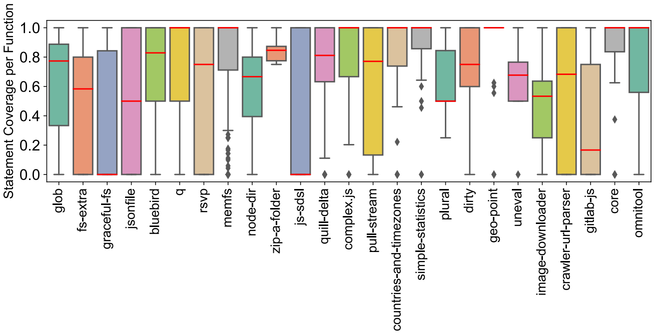

Fig. 5: Distribution of statement coverage per function for TESTPILOT's generated tests using *gpt3.5-turbo*.

Nessie generates 1000 tests, for which we measure statement and branch coverage in the same way as for TESTPILOT. We then repeat these measurements 10 times and take the median coverage across the 10 runs to follow a similar setup to TESTPILOT's evaluation. We use a Wilcoxon paired rank-sum test to determine if there are statistically significant differences between the coverage achieved by both tools.

The last two columns of Table 2 show statement and branch coverage for Nessie. We note that Nessie could not run on `uneval`, because the module's only export is a function, which Nessie does not support. For the remaining 24 packages, Nessie achieved 4.7%– 96.0% statement coverage, with a median of 51.3%. In contrast, as shown in Table 2, TESTPILOT's median statement coverage is much higher at 70.2%. The difference in branch coverage is even higher, with 52.8% for TESTPILOT vs 25.6% for Nessie. Both these differences are statistically significant (p-values 0.002 and 0.027 respectively) with a large effect size, measured by Cliff's delta [55], of 0.493 for statement coverage and a medium one (0.431) for branch coverage.[10] Note that Nessie always generates 1000 tests per package, while TESTPILOT usually generates far fewer tests, except on `memfs` and `omnitool`. It is also worth emphasizing that Nessie (and other test-generation techniques such as LambdaTester [56]) report coverage of *all* generated tests, regardless of whether they pass or fail while our reported coverage numbers are for passing tests only.

We now dive into the results at the package level. For each package, Table 2 highlights the higher coverage from the two techniques in bold. TESTPILOT outperforms Nessie on 17 of the 24 packages (glob, fs-extra, bluebird, q, rsvp, memfs, js-sdsl, quill-delta, complex.js, pull-stream, simple-statistics, plural, dirty, geo-point, image-downloader, core, omnitool), increasing coverage by 3.6%–74.5%, with a median 30.0% increase. For 7 of the remaining packages (graceful-fs, jsonfile, node-dir, zip-a-folder, countries-and-timezones, crawler-url-parser, gitlab-js), TESTPILOT achieves lower coverage than Nessie. For these packages, it reduces coverage by 0.5%– 53.2%, with a median 3.6% decrease. We also note that Nessie fails to achieve any branch coverage on 3 projects (dirty, geo-point, core), while the statement coverage for these projects is non-zero. Upon further examination, and after consulting the

[8] Exact correlation coefficients and p-values are provided in our artifact.

[9] Note that Nessie's implementation has been refactored and improved after the publication of the original paper, which is why some of the values in this table differ slightly from the published numbers. Nessie's first author has kindly helped us run the improved version (specifically, https://github.com/emarteca/nessie/tree/86e48f1d038d98dcd2663d6d36a58a70c4666b1b) on all 25 packages. We include the Nessie results in our artifact

[10] All effect sizes for all statistical tests are available in our artifact.

```
 1 let manuelmhtr_countries_and_timezones = require("../../TEST_REPO_manuelmhtr_countries_and_timezones");
 2 module.exports = function() {
 3         let ret_val_manuelmhtr_countries_and_timezones_1;
 4         try {
 5                 ret_val_manuelmhtr_countries_and_timezones_1 = manuelmhtr_countries_and_timezones.getAllCountries();
 6                 Promise.resolve(ret_val_manuelmhtr_countries_and_timezones_1).catch(e => { console.log({"error_1": true}); });
 7         } catch(e) {
 8                 console.log({"error_1": true});
 9         }
10         let ret_val_manuelmhtr_countries_and_timezones_2;
11         try {
12                 ret_val_manuelmhtr_countries_and_timezones_2 =
13                   manuelmhtr_countries_and_timezones.getCountry({"k": -293.76984807333383,
14                                                                  "rHMR": -17.71151399309167,
15                                                                  "vSF6": 721.0602634375625,
16                                                                  "l": 497.17371230897766,
17                                                                  "EnL": -611.9090030925536});
18                 Promise.resolve(ret_val_manuelmhtr_countries_and_timezones_2).catch(e => { console.log({"error_2": true}); });
19         } catch(e) {
20                 console.log({"error_2": true});
21         }
22 }
```

Fig. 6: Example of a test generated by Nessie. Highlighted lines are for debugging purposes only and do not contribute to the testing of the package under test.

Nessie authors, we found that Nessie cannot generate tests that instantiate classes, meaning that statement coverage is barely above loading coverage for packages with a class-based API, while the branch coverage is zero.

Aside from the difference in coverage achieved by Nessie and TestPilot, tests generated by Nessie tend to look quite different from the ones generated by TestPilot, which stems from Nessie's random approach to test generation. To illustrate this, Figure 6 shows an example of a test generated by Nessie that exercises the `getCountry` function of `countries-and-timezones`. As can be seen in the figure, the test uses long variable names such as `ret_val_manuelmhtr_countries_and_timezones_1` that hamper readability. Moreover, the test invokes `getCountry` on lines 13–17 with an object literal that binds random values to some randomly named properties, which does not reflect intended use of the API. Moreover, tests generated by Nessie do not contain any assertions. By contrast, tests generated by TestPilot for the same package (see Figure 3) typically use variable names that are similar to those chosen by programmers, invoke APIs with sensible values, and often contain assertions.

4.3 RQ3: Non-trivial Assertions

We define a *non-trivial assertion* as an assertion that depends on at least one function from the package under test. To identify non-trivial assertions, we first use CodeQL [57] to compute a backwards program slice from each assertion in the generated tests. We consider assertions whose backwards slice contains an import of the package under test as non-trivial assertions. We then report generated tests that contain at least one non-trivial assertion.

Table 3 shows the number of tests with non-trivial assertions (*non-trivial test* for short) and their proportion w.r.t all generated tests from Table 2. The table also shows the number and proportion of these tests that pass, along with the statement coverage they achieve.

TABLE 3: Number (%) of ***non-trivial*** TestPilot tests generated using *gpt3.5-turbo* and the resulting statement coverage from the passing non-trivial tests.

Project	**Non-trivial Tests (%)**	**Passing Non-trivial Tests**	
		Tests (%)	**Stmt Cov**
glob	37 (54.4%)	3 (8.1%)	50.1%
fs-extra	142 (30.1%)	70 (49.5%)	28.0%
graceful-fs	64 (18.4%)	27 (42.5%)	41.5%
jsonfile	4 (32.0%)	0 (0.0%)	0.0%
bluebird	227 (61.4%)	137 (60.4%)	61.6%
q	235 (72.6%)	136 (58.0%)	66.4%
rsvp	68 (62.4%)	48 (70.6%)	67.6%
memfs	758 (73.1%)	356 (47.0%)	77.4%
node-dir	7 (16.5%)	0 (0.0%)	0.0%
zip-a-folder	1 (9.1%)	0 (0.0%)	0.0%
js-sdsl	349 (85.3%)	44 (12.6%)	33.9%
quill-delta	92 (60.5%)	27 (28.8%)	59.7%
complex.js	190 (90.9%)	104 (54.6%)	62.7%
pull-stream	60 (72.3%)	29 (47.5%)	64.7%
countries-and-timezones	22 (78.6%)	7 (31.8%)	73.5%
simple-statistics	189 (53.6%)	115 (60.6%)	46.9%
plural	12 (92.3%)	8 (66.7%)	73.8%
dirty	29 (41.7%)	13 (44.8%)	66.0%
geo-point	60 (78.9%)	34 (56.7%)	64.6%
uneval	4 (57.1%)	2 (50.0%)	68.8%
image-downloader	0 (0.0%)	–	0.0%
crawler-url-parser	6 (42.9%)	1 (16.7%)	49.5%
gitlab-js	104 (73.8%)	12 (11.5%)	49.3%
core	64 (74.7%)	12 (18.9%)	75.5%
omnitool	977 (94.6%)	319 (32.6%)	73.8%
Median	61.4%	43.7%	61.6%

We observe that there is only one package, `image-downloader` where TestPilot generates only trivial tests. While the generated tests for `image-downloader` did include calls to its API, they were all missing assert statements. Across the remaining packages, a median of 9.1% – 94.6% of TestPilot's generated tests per package are non-trivial. A median of 61.4% of the generated tests for a given package are non-trivial. When compared to all generated tests, we can also see that only a slightly lower proportion of non-trivial tests pass (median 48.0% for overall passing tests from Table 2 vs. 43.7% for non-trivial passing tests from Table 3). Both these results show that TestPilot typically generates tests with assertions that exercise functionality from the target package.

The coverage achieved by the non-trivial tests also supports this finding. Specifically, when comparing the state-

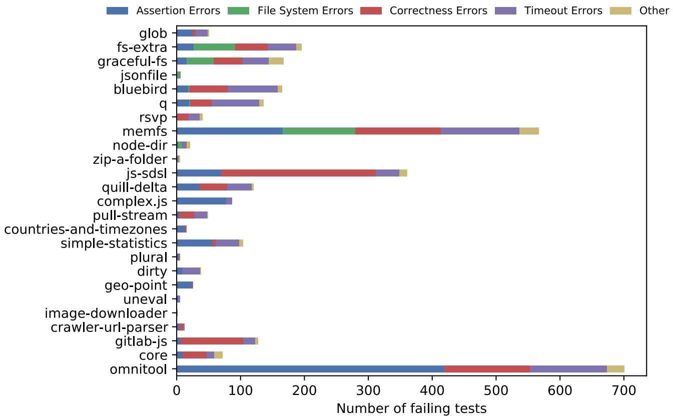

Fig. 7: Types of errors in the failed tests generated by TESTPILOT, using *gpt3.5-turbo*.

ment coverage for all the generated tests in Table 2 to that for non-trivial tests in Table 3, we find that the difference ranges from 0.0%–84.0%, with a median difference of only 7.5%. This means that the achieved coverage for most packages mainly comes from exercising API functionality that is tested by the generated oracles. We note however that there are 4 packages (jsonfile, node-dir, zip-a-folder, image-downloader) where non-trivial tests achieve 0% statement coverage, causing the larger differences. Apart from `image-downloader` discussed above, the three remaining packages do not have any passing non-trivial tests. Since we calculate coverage for passing tests only, this results in the 0% statement coverage for the non-trivial tests.

4.4 RQ4: Characteristics of Failing Tests

Figure 7 shows the number of failing tests for each package, along with the breakdown of the reasons behind the failure. Assertion errors occur when the expected value in an assertion does not match the actual value from executing the code. File-system errors include errors such as files or directories not being found, which we identify by checking for file-system related error codes [58] in the error stack trace. Correctness errors include all type errors, syntax errors, reference errors, incorrect invocations of `done`, and infinite recursion/call stack errors. Timeout errors occur when tests exceed the maximum running time we allow them (2s/test). Finally, we group all other application-specific errors we observe under Other.

We find that the most common failure reason is timeouts with a median 22.7% of failing tests, followed by correctness errors (particularly type errors) with a median of 20.0% of failing tests. The majority of timeouts are due to missing calls to `done`, causing Mocha to keep waiting for the call. We note that on average, the *RetryWithError* refiner was able to fix 15.4% of such timeout errors, with the model often simply adding a call to `done`[11].

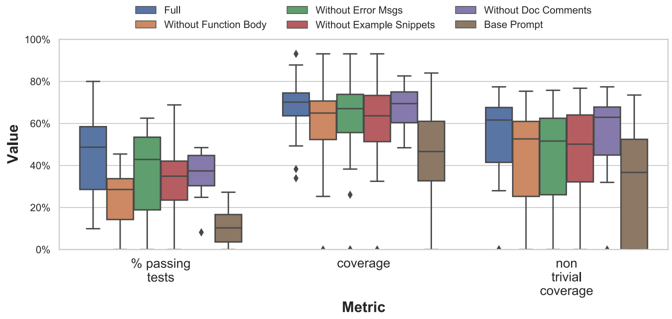

Fig. 8: Effect of disabling prompt refiners in TESTPILOT, using *gpt3.5-turbo*. *Full* considers all refiners while *Base* includes only the function signature and Mocha scaffolding.

We find that a median 19.2% of failures are assertion errors, indicating that in some cases *gpt3.5-turbo* is not able to figure out the correct expected value for the test oracle. This is especially true when the package under test is not widely used and none of the information we provide the model can help it in figuring out the correct values. For example, in one of the tests for `geo-point`, TESTPILOT was able to use coordinates in the provided example snippet to correctly construct two geographical coordinates as input for the `calculateDistance` function, which computes the distance between the two coordinates. However, TESTPILOT incorrectly generated 131.4158102876726 as the expected value for the distance between these two points, while the correct expected value is 130584.05017990958; this caused the test to fail with an assertion error. We note that in this specific case, when TESTPILOT re-prompted the model with the failing test and error message, it was then able to produce a passing test with the corrected oracle. On average across the packages, we find that the *RetryWithError* refiner was able to fix 11.1% of assertion errors.

Finally, we note that file-system errors are domain specific. The generated tests for packages in the file system domain, such as `fs-extra` or `memfs`, have a high proportion of failing tests due to such errors. This is not surprising given that these tests may rely on files that may be non-existent or require containing specific content. Packages in the other domains do not face this problem.

Overall, we find that re-prompting the model with the error message of failing tests (regardless of the failure reason) allows TESTPILOT to produce a consequent passing test in 15.6% of the cases.

4.5 RQ5: Effect of Prompt Refiners

Our results so far include tests generated with all four prompt refiners discussed in Section 2. In this RQ, we investigate the effect of each of these refiners on the quality of the generated tests. Specifically, we conduct an ablation study where we disable one refiner at a time. Disabling a refiner means that we no longer generate prompts that include the information it provides. For example, disabling *DocCommentIncluder* means that none of the prompts we generate would contain documentation comments. We then compare the percentage of passing tests, the achieved coverage, as well as the coverage by non-trivial tests (*non-trivial coverage*).

[11] While the insertion of missing calls to `done` may seem straightforward and therefore be amenable to automated repair, it can be surprisingly tricky to find the correct locations where to insert such calls, and handling this correctly would require applying static analysis to the generated test. We therefore opted for an automated approach that relies solely on the LLM but will consider the use of static analysis to repair generated tests as future work.

```
let mocha = require('mocha');
let assert = require('assert');
let complex_js = require('complex.js');
// complex.js.ZERO.valueOf()
describe('test suite', function() {
    it('test case', function(done) {
        assert.equal(complex_js.ZERO.valueOf(), 0);
        done();
    })
})
```

(a)

```
let mocha = require('mocha');
let assert = require('assert');
let complex_js = require('complex.js');
// complex.js.ZERO.valueOf()
// function() {
//   if (this['im'] === 0) {
//     return this['re'];
//   }
//   return null;
// }
describe('test complex_js', function() {
    it('test complex.js.ZERO.valueOf', function(done) {
      assert.equal(complex_js.ZERO.valueOf(), null);
      done();
    })
})
```

(b)

Fig. 9: Example of a refinement negatively influencing test generation. Prompt (a) contains no information except the method signature, and the generated test passes. Prompt (b) adds the body of the method, but the generated test fails.

Figure 8 shows our results. The x-axis shows the metrics we compare across the different configurations shown in the legend. The y-axis shows the values for each metric (all percentages). Each data point in a boxplot represents the results of the specific metric for a given package, using the corresponding refiner configuration. The black line in the middle of each box represents the median value for each metric across all packages. The full configuration is the configuration we presented so far (i.e., all refiners enabled). The other configurations show the results of excluding only one of the refiners. For example, the red box plot shows the results when disabling the *SnippetIncluder* (i.e., *Without Example Snippets*). The base prompt configuration contains only the function signature and test scaffolding (i.e., disabling all refiners). Note, however, that only 8 of the packages in our evaluation contain documentation comments. It does not make sense to compare the effect of disabling the *DocCommentIncluder* on packages that do not contain doc comments in the first place. Therefore, while the distributions shown in all boxplots represent 25 packages, the *Without Doc Comments* configuration contains data for only 8 packages.

Overall, we can see that the full configuration outperforms all other configurations, across all three metrics, implying that all the prompt information we include contributes to generating more effective tests. We find that there was not a single package where disabling a refiner led to better results on any metric. With the exception of 4 packages where disabling one of the refiners did not affect the results (*SnippetIncluder* on `crawler-url-parser` and `dirty`; and *RetryWithError* on `gitlab-js` and `zip-a-folder`), disabling a refiner *always* resulted in lower values in at least one metric.

The contributions of the refiners are especially notable for the percentage of passing tests where disabling any of the refiners (e.g., *FnBodyIncluder* or *SnippetIncluder*) results in a large drop in the percentage of passing tests. This suggests that the refiners are effective in guiding the model towards generating more passing tests, even if this does not necessarily result in additional coverage. We find that across *all* packages, a full configuration always leads to a higher percentage of passing tests for a given API, while maintaining high coverage.

To understand if the differences between the distributions we observe in Figure 8 are statistically significant, we compare the results of each pair of configurations for all three metrics using a Wilcoxon matched pairs signed rank tests. Note that when comparing against *DocCommentIncluder*, we compare distributions for only the 8 packages that contain doc comments.

We find a statistically significant difference between the full configuration and each configuration that disables *any* refiner as well as between the base configuration and each of the other configurations. Compared to the full configuration, the largest effect size we observed for disabling a refiner was on passing tests when either *FnBodyIncluder* or *DocCommentIncluder* were disabled (Cliff's delta 0.582 and 0.531 respectively).

Apart from differences with the full and base configuration, we find no statistically significant differences between the pairs of other configurations except for the following cases: We find that for both passing tests and coverage, there is a statistically significant difference between the configuration that disables *FnBodyIncluder* and that which disables *RetryWithError* (medium and negligible effect sizes, respectively). For passing tests, we also find a statistically significant difference between disabling *FnBodyIncluder* and disabling each of *SnippetIncluder* and *DocCommentIncluder* (small and medium effect sizes, respectively). However, we note that a sample size of 8 is too small to draw any valid conclusions for *DocCommentIncluder*. It is particularly interesting to see that there was no statistically significant difference between disabling *SnippetIncluder* and disabling any of the other refiners. This suggests that the absence of example snippets does not necessarily affect the metrics any more than the absence of any of the information provided by the other refiners. Since Figure 8 shows that we still obtain a high median coverage even when disabling *SnippetIncluder*, this suggests that the presence of examples snippets is not essential for generating effective test suites with high coverage, and that TestPilot is applicable even in cases where no documentation examples are present.

Finally, we note that while the overall results across a given package show that the refiners always improve, or at least maintain, coverage and percentage of passing tests,

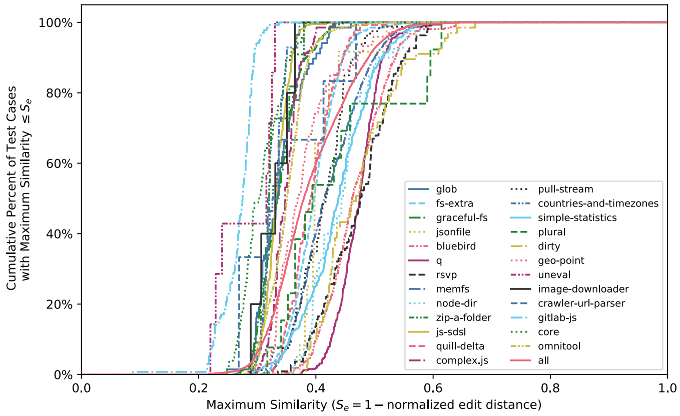

Fig. 10: Cumulative percent of TESTPILOT generated test cases, using *gpt3.5-turbo*, with maximum similarity less than the similarity value shown on the x-axis.

this does not mean that a refiner always improves the results for an individual API function. We have observed situations where adding information such as the function implementation to a prompt that does not include it confuses the model, resulting in the generation of a failing test. Figure 9 shows an example for the `complex.js` package: given the base prompt on the left, *gpt3.5-turbo* is able to produce a (very simple) passing test for the `valueOf` method of the constant `ZERO` exported by the package; adding the function body yields the prompt on the right, which seems to confuse the model, resulting in the generation of a failing test. Across all packages, 5,367 prompts were generated by applying one of the refiners, and in only 394 cases (7.3%) the refined prompt was less effective than the original prompt in the sense that a passing test was generated from the original prompt, but not from the refined prompt.

4.6 RQ6: Memorization

Since *gpt3.5-turbo* was trained on GitHub code, some of the existing tests included in our benchmarks may have been part of its training set. This raises the concern that TESTPILOT may be memorizing existing tests, rather than generating new ones, limiting its usefulness for packages it was not trained on. To investigate potential effects of memorization, we measure the similarity between each generated test and the existing tests in the benchmarks (number of existing tests shown in Table 1). Recently, Lemieux et al. [27] reported that code plagiarism or clone detection [59] techniques are not effective at identifying LLM code memorization. Instead, they find that measuring similarity through edit distance [60] produces more meaningful results. They define *maximum similarity* as a metric that measures the normalized highest similarity between a given generated test and all existing tests as follows: $\max_{tp \in TP}\left(1 - \frac{dist(t^*,t_p)}{\max(len(t^*),len(tp))}\right)$, where TP is the set of existing test functions in a package, t^* is a given generated test, and $dist$ is the edit distance between a generated test and an existing test. We follow the same method for calculating maximum similarity for each generated test, using the npm Levenstein package [61] to calculate $dist$.

```
it('test case', function(done) {
    bluebird.resolve().then(function() {
        throw new Error('test');
    }).catchThrow().catch(function(err) {
        assert.equal(err.message, 'test');
    }).finally(done);
});
```

(a) TESTPILOT generated test case

```
it("1 level", function() {
    return Promise.resolve().then(function() {
        throw new Error();
    }).then(assert.fail, function(e) {
        assertLongTrace(e, 1 + 1, [1]);
    });
});
```

(b) Existing test case with similarity 0.62

Fig. 11: Example of a generated test case from `bluebird`, with max similarity 0.63 to an existing test case

Figure 10 shows the cumulative percentage of generated tests cases for each project where the maximum similarity is less than the value on the x-axis. We also show this cumulative percentage for *all* generated test cases across all projects. We find that 6.2% of TESTPILOT's generated test cases have less than $\leq 0.3\%$ maximum similarity to an existing test, 60.0% have ≤ 0.4 similarity, 92.8% have ≤ 0.5, 99.6% have ≤ 0.6 while 100.0% of the generated tests cases have ≤ 0.7. This means that TESTPILOT **never** generates exact copies of existing tests. In contrast, while 90% of Lemieux et al. [27]'s generated Python tests have ≤ 0.4 similarity, 2% of their test cases are exact copies. That said, given the differences between testing frameworks in Python and JavaScript (e.g., Mocha requires more boilerplate code than pytest), similarity numbers cannot be directly compared between the two languages.

To further illustrate the resulting similarity numbers, Figure 11 shows an example of a test case from `bluebird` with 0.62 similarity to an existing test case. While the edit distance here is low, resulting in the high similarity, we can see that the tests have semantic differences. For example, the generated test simply checks that the thrown exception is a type error, while the existing test checks for certain values in the trace. Thus, the 7.2% of test cases we generate with > 0.5 similarity do not pose a concern that TESTPILOT is generating memorized test cases. Finally, we would expect the generated tests for GitLab-hosted projects to have a lower similarity to existing tests since, as far as we know, the training set for OpenAI's models only includes projects from GitHub, so the model is less likely to have seen the existing tests during training. Our results do indeed show that three out of the five projects have a maximum similarity of ≤ 0.4, with the remaining two having maximum similarity of 0.5. This gives us confidence that the similarity metric we use provides meaningful results.

4.7 RQ7: Effect of Different LLMs

Table 4 shows the number of generated tests, percent of generated tests that pass, as well as statement and branch coverage of TESTPILOT's generated tests when using three different LLMs. While the individual coverage per package varies, we can see that the coverage of tests generated by the *code-cushman-002* model is comparable to those generated

TABLE 4: A comparison of statement coverage of TestPilot's generated tests using three LLMs. For each project, we show the number of generated tests, the number (%) of passing tests, and the statement coverage achieved by these passing tests.

Project	*gpt3.5-turbo*				*code-cushman-002*				*StarCoder*			
	Tests	**Passing**	**Stmt Coverage**	**Branch Coverage**	**Tests**	**Passing**	**Stmt Coverage**	**Branch Coverage**	**Tests**	**Passing**	**Stmt Coverage**	**Branch Coverage**
glob	68	18 (26.5%)	**71.3%**	**66.3%**	76	31 (40.1%)	61.7%	51.0%	45	10 (22.2%)	64.8%	58.4%
fs-extra	471	277 (58.8%)	**58.8%**	**38.9%**	394	254 (64.3%)	41.0%	23.3%	443	163 (36.7%)	43.0%	25.5%
graceful-fs	345	177 (51.4%)	**49.3%**	**33.3%**	301	135 (44.9%)	47.5%	30.3%	309	100 (32.4%)	44.7%	22.7%
jsonfile	13	6 (48.0%)	38.3%	29.4%	15	8 (53.3%)	46.8%	44.1%	13	7 (53.8%)	**59.6%**	**47.0%**
bluebird	370	204 (55.2%)	68.0%	50.0%	400	211 (52.6%)	**68.2%**	**51.3%**	395	130 (32.8%)	55.6%	36.0%
q	323	186 (57.6%)	**70.4%**	**53.7%**	356	190 (53.4%)	66.9%	51.2%	348	96 (27.4%)	63.0%	48.1%
rsvp	109	70 (64.2%)	70.1%	55.3%	115	77 (66.5%)	**73.3%**	**60.5%**	141	45 (31.9%)	66.8%	53.2%
memfs	1037	471 (45.4%)	**81.1%**	**58.9%**	1058	505 (47.7%)	78.9%	54.9%	922	268 (29.0%)	71.9%	49.8%
node-dir	40	19 (48.1%)	**64.3%**	**50.8%**	22	16 (74.4%)	52.2%	41.1%	51	17 (33.3%)	54.0%	42.7%
zip-a-folder	11	6 (54.5%)	84.0%	50.0%	10	7 (70.0%)	**88.0%**	**62.5%**	11	4 (36.4%)	56.0%	37.5%
js-sdsl	409	46 (11.3%)	33.9%	24.3%	274	63 (23.0%)	**36.5%**	**27.3%**	235	21 (8.9%)	26.9%	17.9%
quill-delta	152	33 (21.7%)	73.0%	64.3%	187	50 (26.5%)	**74.0%**	**66.6%**	135	7 (5.2%)	31.0%	21.1%
complex.js	209	121 (58.0%)	**70.2%**	**46.5%**	221	125 (56.3%)	62.7%	46.2%	178	56 (31.5%)	53.5%	34.9%
pull-stream	83	34 (41.0%)	69.1%	52.8%	76	43 (55.9%)	**70.8%**	**54.7%**	69	10 (14.5%)	51.6%	32.7%
countries-and-timezones	28	13 (46.4%)	**93.1%**	69.1%	41	18 (44.4%)	**93.1%**	**74.4%**	33	11 (33.8%)	88.2%	64.9%
simple-statistics	353	250 (70.9%)	**87.8%**	**71.3%**	350	213 (60.7%)	80.1%	63.9%	352	164 (46.6%)	69.9%	54.5%
plural	13	8 (61.5%)	73.8%	**59.1%**	17	8 (47.1%)	**75.4%**	**59.1%**	13	5 (38.5%)	73.8%	**59.1%**
dirty	70	32 (45.3%)	74.5%	65.4%	89	42 (47.5%)	**81.1%**	**69.2%**	57	23 (40.4%)	72.6%	61.5%
geo-point	76	50 (65.8%)	**87.8%**	**70.6%**	87	35 (40.2%)	61.0%	**70.6%**	62	16 (25.8%)	46.3%	**70.6%**
uneval	7	2 (28.6%)	**68.8%**	**58.3%**	5	0 (0.0%)	0.0%	0.0%	6	0 (0.0%)	0.0%	0.0%
image-downloader	5	4 (80.0%)	63.6%	**50.0%**	5	2 (40.0%)	**75.8%**	**50.0%**	5	2 (40.0%)	63.6%	**50.0%**
crawler-url-parser	14	2 (14.3%)	**51.4%**	**35.0%**	17	2 (11.8%)	49.5%	31.3%	14	1 (7.1%)	48.6%	32.5%
gitlab-js	141	14 (9.9%)	51.7%	16.5%	116	35 (29.7%)	**61.8%**	**31.8%**	117	1 (0.9%)	28.4%	0.6%
core	85	13 (15.3%)	**78.3%**	**50.0%**	102	21 (20.7%)	72.7%	47.7%	61	5 (8.2%)	16.1%	0.0%
omnitool	1033	330 (31.9%)	**74.2%**	**55.2%**	1029	321 (31.1%)	70.1%	54.2%	812	194 (23.9%)	40.0%	18.1%
Median		**48.0%**	**70.2%**	**52.8%**		47.1%	68.2%	51.2%		31.5%	54.0%	37.5%

by *gpt3.5-turbo*, with the latter having a slightly higher median statement and branch coverage across the packages. A Wilcoxon matched-pairs signed-rank test shows no statistically significant differences between *gpt3.5-turbo* and *code-cushman-002* for either type of coverage. On the other hand, we do find a statistically significant difference between *StarCoder* and each of the OpenAI models (p-value < 0.05) for both types of coverage. As shown in Table 4, *StarCoder* achieves lower median statement (54.0%) and branch coverage (37.5%) than both other models. Cliff's delta [55] shows a large and medium effect size for statement and branch coverage, respectively, between *gpt3.5-turbo* and *StarCoder* and a medium and small effect size for statement and branch coverage, respectively between *code-cushman-002* and *StarCoder*.

However, we note that *StarCoder*'s median statement coverage and branch coverage are both higher than Nessie (statement: 54.0% vs. 51.3% and branch: 37.5% vs 25.6%). While this higher coverage was not statistically significant, the results show that even LLMs trained with potentially smaller datasets and/or a different training process than OpenAI's models are on par (or even sometimes higher) than state-of-the-art traditional test-generation techniques, such as Nessie [11]. Furthermore, in **RQ2**, we showed that using *gpt3.5-turbo* with TestPilot resulted in higher coverage test suites, with statistically significant differences to Nessie. Overall, these results emphasize the promise of LLM-based test generation techniques in generating high coverage test suites.

Finally, we note that the median time for TestPilot to generate tests for a given function using *gpt3.5-turbo* is 15s, and the median time to generate a complete test suite for a given package is 6m 55s.[12] The bulk of this time is spent querying the model, so the choice of LLM makes a big difference. For example, the median time for TestPilot to generate tests for a given function using *StarCoder* and *code-cushman-002* is 24s and 11s, respectively, and 10m 48s and 4m 53s, respectively, for a complete test suite. All these performance numbers suggest that it is feasible to use TestPilot either in an online setting (e.g., in an IDE) to generate tests for individual functions, or in an offline setting (e.g., during code review) to generate complete test suites for an API.

5 Threats to Validity

Internal Validity: The extraction of example snippets from documentation relies on textually matching a function's name. Given two functions with the same name but different access paths, we cannot disambiguate which function is being used in the example snippet. Accordingly, we match this snippet to both functions. While this may lead to inaccuracies, there is unfortunately no precise alternative for this matching. Any heuristics may cause us to miss snippets altogether, which may be worse since example snippets help with increasing the percentage of passing tests as shown in Figure 8. The overall high coverage and percentage of passing tests suggest that our matching technique is not a limiting factor in practice.

Construct Validity: We use the concept of non-trivial assertions as a proxy for oracle quality in the generated tests. When determining non-trivial assertions, we search for *any* usage of the package under test in the backwards slice of the assertion. Such usage may be different from the intended function under test. However, given the dynamic nature of JavaScript, precisely determining the usage of a given function, as extracted by the API explorer, and its occurrence in the backwards slice is difficult. While our approach does not allow us to precisely determine non-trivial coverage for a given function, this does not affect the non-trivial coverage we report for each package's complete API. Note that when calculating non-trivial coverage, we measure the full coverage of tests that contain *at least* one non-trivial assertion. There may be other calls in those non-trivial tests that contribute to coverage but do not contribute to the assertion. Measuring assertion/checked coverage as defined by Schuler and Zeller [62] is a possible alternative, but this is practically difficult to implement precisely for JavaScript.

Our definition of non-trivial assertions is simple, setting a low bar for non-triviality. Any assertion classified as trivial

[12] These timings were measured on a standard GitHub Actions Linux VM with a 2-core CPU, 7GB of RAM, and 14GB of SSD disk space.

by our criterion is, indeed, not meaningful, but the converse is not necessarily true. Accordingly, our measure of non-trivial coverage is a lower bound on the true non-trivial coverage.

While the examples we show in the paper suggest that TESTPILOT's generated tests use variable names that are similar to those chosen by programmers, we do not formally assess the readability of these tests. In the future, it would be interesting to conduct user studies to assess the readability of tests generated by different techniques.

External Validity: Despite our evaluation scale significantly exceeding evaluations of previous test generation approaches [11], [25], our results are still based on 25 npm packages and may not generalize to other JavaScript code bases. In particular, TESTPILOT's performance may not generalize to proprietary code that was never seen in the LLM's training set. We try to mitigate this effect in several ways: (1) we evaluate on less popular packages that are likely to have had less impact on the model's training, (2) we evaluate on 5 GitLab repositories that have not been included in the models' training, and (3) we measure the similarity of the generated tests to the existing tests. Our results show that TESTPILOT performs well for both popular and unpopular packages and that 92.8% of the test cases have $\leq$ 50% similarity with existing tests, with no exact copies. Overall, this reassures us that TESTPILOT is not producing "memorized" code.

Finally, we note that while our technique is conceptually language-agnostic, our current implementation of TESTPILOT targets JavaScript, and thus we cannot generalize our results to other languages.

6 RELATED WORK

TESTPILOT provides an alternative to (and potentially complements) traditional techniques for automated test generation, including feedback-directed random test generation [7]–[11], search-based and evolutionary techniques [16], [17], [63], [64], and dynamic symbolic execution [12]–[15]. This section reviews neural techniques for test generation, and previous test generation techniques for JavaScript.

6.1 Neural Techniques

Neural techniques are rapidly being adopted for solving various Software Engineering problems, with promising results in several domains including code completion [34]–[38], program repair [39]–[41], and bug-finding [42], [43]. Pradel and Chandra [65] survey the current state of the art in this emerging research area. We are aware of several recent research efforts in which LLMs are used for test generation [23]–[29]. There are two main differences between our work and these efforts: (i) the goal and types of tests generated and (ii) the need for some form of fine-tuning or additional data. We discuss the details below.

Differing goals: TICODER [24] and CODET [23] use Codex to generate implementations and test cases from problem descriptions expressed in natural language. TICODER relies on a test-driven user-intent formalization (TDUIF) loop in which the user and model interact to generate both an implementation matching the user's intent and a set of test cases to validate its correctness. CODET, on the other hand, generates both a set of candidate implementations and some test cases based on the same prompt, runs the generated tests on the candidate implementations, and chooses the best solution based on the test results. Unlike TESTPILOT, neither of these efforts solves the problem of automatically generating unit tests for *existing code*.

Given the characteristics of LLMs in generating natural looking code, there have been several efforts exploring the use of LLMs to help [27] or complement [28] traditional test generation techniques. Most recently, Lemieux et al. [27] explore using tests generated by Codex as a way to unblock the search process of test generation using search-based techniques [64], which often fails when the initial randomly generated test has meaningless input that cannot be mutated effectively. Their results show that, on most of their target 27 Python projects, their proposed technique, CODAMOSA, outperforms the baseline search-based technique, Pynguin's implementation of MOSA [64], as well as using only Codex. However, their Codex prompt includes only the function implementation and an instruction to generate tests. Since their main goal is to explore whether a test generated by Codex can improve the search process, they do not systematically explore the effect of different prompt components. In fact, they conjecture that further prompt engineering might improve results, motivating the need for our work which systematically explores different prompt components. Additionally, their generated tests are in the MOSA format [64], which the authors acknowledge could lose readability, and do not contain assertions. Most of our tests contain assertions, and we further study the quality of assertions we generate as well as reasons for test failures.

Similarly, given that it is often difficult for traditional test generation techniques to generate (useful) assertions [21], [22], ATLAS [28] uses LLMs to generate an assert statement for a given (assertion-less) Java test. They position their technique as a complement to traditional techniques [8], [17]. With the same goal, Mastrapaolo et al. [29], [45] and Tufano et al. [46] perform follow up work using transfer learning, while Yu et al. [66] use information retrieval techniques to further improve the assert statements generated by Atlas. TOGA [67] uses similar techniques but additionally incorporates an exceptional oracle classifier to decide if a given method requires an assertion to test exceptional behavior. It then bases the generation of the assertion on a pre-defined oracle taxonomy created by manually analyzing existing Java tests and using a neural-based ranking mechanism to rank candidates with oracles higher. In contrast with these efforts, our goal is to generate a *complete* test method without giving the model any content of the test method (aside from boilerplate code required by Mocha), which means that the model needs to generate *both* any test setup code (e.g., initializing objects and populating them) as well as the assertion. While TOGA can be integrated with EvoSuite [16] to create an end-to-end test-generation tool, recent work [68] points out several shortcomings of the evaluation methods, casting doubt on the validity of the reported results.

Differing Input/Training: Bareiß et al. [25] evaluate the performance of Codex on three code-generation tasks, including test generation. Like us, they rely on embedding contextual information into the prompt to guide the LLM, though the specific data they embed is different: while TESTPILOT only includes the signature, definition, documentation, and usage examples in the prompt, Bareiß et al. pursue a few-shot learning approach where, in addition to the definition of a function under test, they include an example of a different function from the same code base and its associated test to give the model a hint as to what it is expected to do, as well as a list of related helper function signatures that could be useful for test generation. For a limited list of 18 Java methods, they show that this approach yields slightly better coverage than Randoop [8], [9], a popular technique for feedback-directed random test generation. This is a promising result, but finding suitable example tests to use in few-shot learning can be difficult, especially since their evaluation shows that good coverage crucially depends on the examples being closely related to the function under test.

Tufano et al. [26] present AthenaTest, an approach for automated test generation based on a BART transformer model [44]. For a given test case, they rely on heuristics to identify the "focal" class and method under test. These mapped test cases are then used to fine-tune the model for the task of producing unit tests by representing this task as a translation task that maps a focal method (along with the focal class, constructors, and other public methods and fields in that class) to a test case. In experiments on 5 projects from Defects4J [69], AthenaTest generated 158K test cases, achieving similar test coverage as EvoSuite [16], a popular search-based test generation tool, and covering 43% of all focal methods. A significant difference between their work and ours is that their approach requires training the model on a large set of test cases whereas TESTPILOT uses an off-the-shelf LLM. In fact, in addition to the goal differences with ATLAS [28] and Mastrapaolo et al.'s [29], [45] work above, both these efforts also require a data set of test methods (with assertions) and their corresponding focal methods, whether to use in the main training [28] or in fine tuning during transfer learning [29], [45], [46].

Unfortunately, the above differences in goals or in the required data for model training make it meaningless or impossible to do a direct experimental comparison with TESTPILOT. Additionally, none of these efforts support JavaScript or provide JavaScript data sets that can be used for comparison. In fact, one of our main motivations for exploring prompt engineering for an off-the-shelf LLM is to avoid the need to collect test examples for few-shot learning [25] or test method/focal method pairs required for training [28] or additional fine tuning [29], [45], [46].

Other techniques: Stallenberg et al. [70] present a test generation technique for JavaScript based on unsupervised type inference consisting of three phases. First, a static analysis is performed to deduce relationships between program elements such as variables and expressions. Then, a probabilistic type inference is applied to these relationships to construct a model. Finally, they show how search-based techniques can take advantage of the information contained in such models by proposing two strategies for consulting these models in the main loop of DynaMOSA [64].

Recently, El Haji [71] presented an empirical study that explores the effectiveness of GitHub Copilot at generating tests. In this study, tests are selected from existing test suites associated with 7 open-source Python projects. After removing the body of each test function, Copilot is asked to complete the implementation so that the resulting test can be executed and compared against the original test. Two variations of this approach are explored, viz., "with context" where the other tests in the suite are preserved and "without context" where other tests are removed. El Haji also explores the impact of (manually) adding comments that include descriptions of intended behavior and usage examples. The results from the study show that 45.28% of generated test are passing in the "with context" scenario (the rest are failing, syntactically invalid, or empty) vs only 7.55% passing generated tests in the "without context" scenario, and that the addition of usage examples and comments is generally helpful. There are several significant differences between our approach and El Haji's work: we explore a fully automated technique without any manual steps, we report on a significantly more extensive empirical evaluation, we present an adaptive technique in which prompts are refined in response to the execution behavior of previously executed tests, we target a different programming language (JavaScript instead of Python), and TestPilot interacts directly with an LLM rather than relying on Copilot, an LLM-based programming assistant.

6.2 Test Generation Techniques for JavaScript

TESTPILOT's mechanism for refining prompts based on execution feedback was inspired by the mechanism employed by feedback-directed random test generation techniques [7]–[11], where new tests are generated by extending previously generated passing tests. As reported in Section 4.2, TESTPILOT achieves significantly higher statement coverage and branch coverage than Nessie [11], which represents the state-of-the-art in feedback-directed random test generation for JavaScript.

Several previous projects have considered test generation for JavaScript (see [72] for a survey). Saxena et al. [73] present Kudzu, a tool that aims to find injection vulnerabilities in client-side JavaScript applications by exploring an application's input space. They differentiate an application's input space into an *event space*, which concerns the order in which event handlers execute (e.g., as a result of buttons being clicked), and a *value space* which concerns the choice of values passed to functions or entered into text fields. Kudzu uses dynamic symbolic execution to explore the value space systematically, but it relies on a random exploration strategy to explore the event space. Artemis [74] is a framework for automated test generation that iteratively generates tests for client-side JavaScript applications consisting of sequences of events, using a heuristics-based strategy that considers the locations read and written by each event handler to focus on the generation of tests involving event handlers that interact with each other. Li et al. [75] extends Artemis with dynamic symbolic execution to improve its ability to explore the value space, and Tanida et al. [76] further improve on this work by augmenting generated test inputs with user-supplied invariants. Fard et al. [77] present ConFix, a tool that uses a combination of dynamic analysis and symbolic execution to automatically generate instances of the Document Object Model (DOM) that can serve as test fixtures in unit tests for client-side JavaScript code. Marchetto and Tonella [78] present a search-based test generation technique that constructs tests consisting of sequences of events that relies on the automatic extraction of a finite state machine that represents that application's state. None of these tools generate tests that contain assertions.

Several test generation tools for JavaScript are capable of generating tests containing assertions. JSART [79] is a tool that generates regression tests that contain assertions reflecting likely invariants that are generated using a variation of the Daikon dynamic invariant generator [80]. Since Daikon generates assertions that are *likely* to hold, an additional step is needed in which invalid assertions are removed from the generated tests. Mirshokraie et al. [81], [82] present an approach in which tests are generated for client-side JavaScript applications consisting of sequences of events. Then, in an additional step, function-level unit tests are derived by instrumenting program execution to monitor the state of parameters, global variables, and the DOM upon entry and exit to functions to obtain values with which functions are to be invoked. Assertions are added automatically to the generated tests by: (i) mutating the DOM and the code of the application under test, (ii) executing generated tests to determine how application state is impacted by mutations, and (iii) adding assertions to the tests that reflect the behavior prior to the mutation. Testilizer [83] is a test generation tool that aims to enhance an existing human-written test suite. To this end, Testilizer instruments code to observe how existing tests access the DOM, and executes them to obtain a State-Flow Graph in which the nodes reflect dynamic DOM states and edges reflect the event-driven transitions between these states. Alternative paths are explored by exploring previously unexplored events in each state. Testilizer adds assertions to the generated tests that are either copied verbatim from existing tests, by adapting the structure of an existing assertion to a newly explored state, or by inferring a similar assertion using machine learning techniques.

These techniques share the limitation that they require the entire application under the test to be executable, limiting their applicability. Moreover, several of the techniques discussed above require re-execution of tests (to infer assertions using mutation testing [81], [82], or to filter out assertions that are invalid [79]), which adds to their cost. By contrast, TESTPILOT only requires the functions of API functions under test to be available and executable, and it executes each test that it generates only once.

7 CONCLUSIONS AND FUTURE WORK

We have presented TESTPILOT, an approach for adaptive unit-test generation using a large language model. Unlike previous work in this area, TESTPILOT requires neither fine tuning nor a parallel corpus of functions and tests. Instead, we embed contextual information about the function under test into the prompt, specifically its signature, its attached documentation comment (if any), any usage examples from the project documentation, and the source code of the function. Furthermore, if a generated test fails, we adaptively create a new prompt embedding this test and the failure message to guide the model towards fixing the problematic test. We have implemented our approach for JavaScript on top of off-the-shelf LLMs, and shown that it achieves state-of-the art statement coverage on 25 npm packages. Further evaluation shows that the majority of the generated tests contain non-trivial assertions, and that all parts of the information included in the prompt contributes to the quality of the generated tests. Experiments with three LLMs (*gpt3.5-turbo*, *code-cushman-002*, and *StarCoder*) demonstrate that LLM-based test generation already outperforms state-of-the-art previous test generation methods such as Nessie on key metrics. We conjecture that the use of more advanced LLMs will further improve results, though we are reluctant to speculate by how much.

In future work, we plan to further investigate the quality of the tests generated by TESTPILOT. While in this paper we have focused on passing tests and excluded failing tests from consideration entirely, we have seen examples of failing tests that are "almost right" and might be interesting to a developer as a starting point for further refinement. However, doing this depends on having a good strategy for telling apart useful failing tests from useless ones. Our experiments have demonstrated that timeout errors, assertion errors, and correctness errors are key factors that cause tests to fail. In future work, we plan to apply static and dynamic program analysis to failing tests in order to determine why timeout errors and assertion errors occur and how failing tests could be modified to make them pass.

Further research is needed to determine what factors prevent the generation of non-trivial assertions. Anecdotally, we have observed that the availability of usage examples is generally helpful. We envision that the number of useful assertions could be improved by obtaining usage examples in other ways, e.g., by interacting with a user, or by extracting usage examples from clients of the application under test.

Another fruitful area of experimentation could be varying the sampling temperature of the LLM. In this work, we always sample at temperature zero, which has the advantage of providing stable results, but also means that the model is less likely to offer lower-probability completions that might result in more interesting tests.

Another area of future work is the development of hybrid techniques that combine existing feedback-directed test generation techniques with an LLM-based technique such as TESTPILOT. For example, one could use an LLM-based technique to generate an initial set of tests and use the tests that it generates as a starting point for extension by a feedback-directed technique such as Nessie, thus enabling it to uncover edges cases that would be difficult to uncover using only random values.

In principle, our approach can be adapted to any programming language. Practically speaking, this would involve adapting prompts to use the syntax of the language under consideration, and to use a testing framework for that language. In addition, the mining of documentation and usage examples would need to be adapted to match the documentation format used for that language. The LLMs that we used did not language-specific training and could be used to generate tests for other languages, though the effectiveness of the approach will depend on the amount of code written in that language that was included in the LLM's training set. One specific question that would be interesting to explore is how well an approach like TESTPILOT would perform on a statically-typed language.

ACKNOWLEDGMENT

The research reported on in this paper was conducted while S. Nadi and F. Tip were sabbatical visitors and A. Eghbali an intern at GitHub. The authors are grateful to the GitHub Next team for many insightful and helpful discussions about TestPilot. F. Tip was supported in part by National Science Foundation grants CCF-1907727 and CCF-2307742. S. Nadi's research is supported by the Canada Research Chairs Program and the Natural Sciences and Engineering Research Council of Canada (NSERC), RGPIN-2017-04289.

REFERENCES

[1] K. Beck, *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.

[2] J. Shore and S. Warden, *The Art of Agile Development*, 2nd ed. O'Reilly, 2021.

[3] S. Siddiqui, *Learning Test-Driven Development*. O'Reilly, 2021.

[4] E. Daka and G. Fraser, "A survey on unit testing practices and problems," in *2014 IEEE 25th International Symposium on Software Reliability Engineering*, 2014, pp. 201–211.

[5] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, 1990. [Online]. Available: https://doi.org/10.1145/96267.96279

[6] M. Zalewski, "American fuzzy lop," https://lcamtuf.coredump.cx/afl/, 2022.

[7] C. Csallner and Y. Smaragdakis, "JCrasher: an automatic robustness tester for Java," *Software Practice and Experience*, vol. 34, no. 11, pp. 1025–1050, 2004.

[8] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for java," in *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, ser. OOPSLA '07. New York, NY, USA: ACM, 2007, pp. 815–816. [Online]. Available: http://doi.acm.org/10.1145/1297846.1297902

[9] C. Pacheco, S. K. Lahiri, and T. Ball, "Finding errors in .net with feedback-directed random testing," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ser. ISSTA '08. New York, NY, USA: ACM, 2008, pp. 87–96. [Online]. Available: http://doi.acm.org/10.1145/1390630.1390643

[10] M. Selakovic, M. Pradel, R. Karim, and F. Tip, "Test generation for higher-order functions in dynamic languages," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 161:1–161:27, 2018. [Online]. Available: https://doi.org/10.1145/3276531

[11] E. Arteca, S. Harner, M. Pradel, and F. Tip, "Nessie: Automatically testing javascript apis with asynchronous callbacks," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 1494–1505. [Online]. Available: https://doi.org/10.1145/3510003.3510106

[12] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, V. Sarkar and M. W. Hall, Eds. ACM, 2005, pp. 213–223. [Online]. Available: https://doi.org/10.1145/1065010.1065036

[13] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, M. Wermelinger and H. C. Gall, Eds. ACM, 2005, pp. 263–272. [Online]. Available: https://doi.org/10.1145/1081706.1081750

[14] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: automatically generating inputs of death," in *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, October 30 - November 3, 2006*, A. Juels, R. N. Wright, and S. D. C. di Vimercati, Eds. ACM, 2006, pp. 322–335. [Online]. Available: https://doi.org/10.1145/1180405.1180445

[15] N. Tillmann, J. de Halleux, and T. Xie, "Transferring an automated test generation tool to practice: from pex to fakes and code digger," in *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, I. Crnkovic, M. Chechik, and P. Grünbacher, Eds. ACM, 2014, pp. 385–396. [Online]. Available: https://doi.org/10.1145/2642937.2642941

[16] G. Fraser and A. Arcuri, "Evolutionary generation of whole test suites," in *Proceedings of the 11th International Conference on Quality Software, QSIC 2011, Madrid, Spain, July 13-14, 2011*, M. Núñez, R. M. Hierons, and M. G. Merayo, Eds. IEEE Computer Society, 2011, pp. 31–40. [Online]. Available: https://doi.org/10.1109/QSIC.2011.19

[17] ——, "EvoSuite: automatic test suite generation for object-oriented software," in *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, T. Gyimóthy and A. Zeller, Eds. ACM, 2011, pp. 416–419. [Online]. Available: https://doi.org/10.1145/2025113.2025179

[18] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds, "An industrial evaluation of unit test generation: Finding real faults in a financial application," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 2017, pp. 263–272.

[19] G. Grano, S. Scalabrino, H. C. Gall, and R. Oliveto, "An empirical investigation on the readability of manual and generated test cases," in *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 2018, pp. 348–3483.

[20] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer, "Modeling readability to improve unit tests," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, E. D. Nitto, M. Harman, and P. Heymans, Eds. ACM, 2015, pp. 107–118. [Online]. Available: https://doi.org/10.1145/2786805.2786838

[21] A. Panichella, S. Panichella, G. Fraser, A. A. Sawant, and V. J. Hellendoorn, "Revisiting test smells in automatically generated tests: Limitations, pitfalls, and opportunities," in *IEEE International Conference on Software Maintenance and Evolution, ICSME 2020, Adelaide, Australia, September 28 - October 2, 2020*. IEEE, 2020, pp. 523–533. [Online]. Available: https://doi.org/10.1109/ICSME46990.2020.00056

[22] F. Palomba, D. D. Nucci, A. Panichella, R. Oliveto, and A. D. Lucia, "On the diffusion of test smells in automatically generated test code: an empirical study," in *Proceedings of the 9th International Workshop on Search-Based Software Testing, SBST@ICSE 2016, Austin, Texas, USA, May 14-22, 2016*. ACM, 2016, pp. 5–14. [Online]. Available: https://doi.org/10.1145/2897010.2897016

[23] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J.-G. Lou, and W. Chen, "CodeT: Code Generation with Generated Tests," 2022. [Online]. Available: https://arxiv.org/abs/2207.10397

[24] S. K. Lahiri, A. Naik, G. Sakkas, P. Choudhury, C. von Veh, M. Musuvathi, J. P. Inala, C. Wang, and J. Gao, "Interactive Code Generation via Test-Driven User-Intent Formalization," 2022. [Online]. Available: https://arxiv.org/abs/2208.05950

[25] P. Bareiß, B. Souza, M. d'Amorim, and M. Pradel, "Code Generation Tools (Almost) for Free? A Study of Few-Shot, Pre-Trained Language Models on Code," in *Proceedings of the 37th ACM/IEEE International Conference on Automated Software Engineering*, 2022.

[26] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, and N. Sundaresan, "Unit test case generation with transformers and focal context," arXiv, May 2021. [Online]. Available: https://www.microsoft.com/en-us/research/publication/unit-test-case-generation-with-transformers-and-focal-context/

[27] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "CodaMOSA: Escaping coverage plateaus in test generation with pre-trained large language models," in *45th International Conference on Software Engineering*, ser. ICSE, 2023.

[28] C. Watson, M. Tufano, K. Moran, G. Bavota, and D. Poshyvanyk, "On Learning Meaningful Assert Statements for Unit Test Cases," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ACM, jun 2020. [Online]. Available: https://doi.org/10.1145%2F3377811.3380429

[29] A. Mastropaolo, S. Scalabrino, N. Cooper, D. Nader Palacio, D. Poshyvanyk, R. Oliveto, and G. Bavota, "Studying the usage of text-to-text transfer transformer to support code-related tasks,"

in *43rd IEEE/ACM International Conference on Software Engineering (ICSE)*, 2021, pp. 336–347.

[30] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[31] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020. [Online]. Available: https://arxiv.org/abs/2005.14165

[32] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating Large Language Models Trained on Code," *CoRR*, vol. abs/2107.03374, 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[33] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago, T. Hubert, P. Choy, C. d. M. d'Autume, I. Babuschkin, X. Chen, P.-S. Huang, J. Welbl, S. Gowal, A. Cherepanov, J. Molloy, D. J. Mankowitz, E. S. Robson, P. Kohli, N. de Freitas, K. Kavukcuoglu, and O. Vinyals, "Competition-level code generation with alphacode," 2022. [Online]. Available: https://arxiv.org/abs/2203.07814

[34] J. Li, Y. Wang, M. R. Lyu, and I. King, "Code completion with neural attention and pointer networks," in *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, J. Lang, Ed. ijcai.org, 2018, pp. 4159–4165. [Online]. Available: https://doi.org/10.24963/ijcai.2018/578

[35] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: code generation using transformer," in *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, P. Devanbu, M. B. Cohen, and T. Zimmermann, Eds. ACM, 2020, pp. 1433–1443. [Online]. Available: https://doi.org/10.1145/3368089.3417058

[36] R. Karampatsis, H. Babii, R. Robbes, C. Sutton, and A. Janes, "Big code != big vocabulary: open-vocabulary models for source code," in *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, G. Rothermel and D. Bae, Eds. ACM, 2020, pp. 1073–1085. [Online]. Available: https://doi.org/10.1145/3377811.3380342

[37] S. Kim, J. Zhao, Y. Tian, and S. Chandra, "Code prediction by feeding trees to transformers," in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 150–162. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00026

[38] GitHub, "GitHub Copilot," https://copilot.github.com/, 2022.

[39] R. Gupta, S. Pal, A. Kanade, and S. K. Shevade, "Deepfix: Fixing common C language errors by deep learning," in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*, S. Singh and S. Markovitch, Eds. AAAI Press, 2017, pp. 1345–1351. [Online]. Available: http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14603

[40] V. J. Hellendoorn, C. Sutton, R. Singh, P. Maniatis, and D. Bieber, "Global relational models of source code," in *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. [Online]. Available: https://openreview.net/forum?id=B1lnbRNtwr

[41] M. Allamanis, H. Jackson-Flux, and M. Brockschmidt, "Self-supervised bug detection and repair," in *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, M. Ranzato, A. Beygelzimer, Y. N. Dauphin, P. Liang, and J. W. Vaughan, Eds., 2021, pp. 27 865–27 876. [Online]. Available: https://proceedings.neurips.cc/paper/2021/hash/ea96efc03b9a050d895110db8c4af057-Abstract.html

[42] M. Pradel and K. Sen, "Deepbugs: a learning approach to name-based bug detection," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 147:1–147:25, 2018. [Online]. Available: https://doi.org/10.1145/3276517

[43] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. [Online]. Available: https://openreview.net/forum?id=BJOFETxR-

[44] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," *arXiv preprint arXiv:1910.13461*, 2019.

[45] A. Mastropaolo, N. Cooper, D. N. Palacio, S. Scalabrino, D. Poshyvanyk, R. Oliveto, and G. Bavota, "Using transfer learning for code-related tasks," *IEEE Transactions on Software Engineering*, 2022.

[46] M. Tufano, D. Drain, A. Svyatkovskiy, and N. Sundaresan, "Generating accurate assert statements for unit test cases using pretrained transformers," in *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*, ser. AST '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 54–64. [Online]. Available: https://doi.org/10.1145/3524481.3527220

[47] L. Reynolds and K. McDonell, "Prompt programming for large language models: Beyond the few-shot paradigm," 2021. [Online]. Available: https://arxiv.org/abs/2102.07350

[48] OpenAI, "OpenAI LLMs: Deprecations," https://platform.openai.com/docs/deprecations, 2023.

[49] HuggingFace, "Starcoder: A state-of-the-art LLM for code," https://huggingface.co/blog/starcoder, 2023.

[50] "Mocha," https://mochajs.org/, 2022.

[51] G. Mezzetti, A. Møller, and M. T. Torp, "Type Regression Testing to Detect Breaking Changes in Node.js Libraries," in *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*, ser. LIPIcs, T. D. Millstein, Ed., vol. 109. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018, pp. 7:1–7:24.

[52] OpenAI, "OpenAI Codex," https://openai.com/blog/openai-codex, 2023.

[53] "Istanbul coverage tool," https://istanbul.js.org/, 2022.

[54] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software testing, verification and reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[55] N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions." *Psychological bulletin*, vol. 114, no. 3, p. 494, 1993.

[56] M. Selakovic, M. Pradel, R. Karim, and F. Tip, "Test generation for higher-order functions in dynamic languages," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–27, 2018.

[57] GitHub, "CodeQL," https://codeql.github.com/, 2022.

[58] O. Foundation, "Node.js error codes," https://nodejs.org/dist/latest-v18.x/docs/api/errors.html#nodejs-error-codes, 2022.

[59] S. Schleimer, D. S. Wilkerson, and A. Aiken, "Winnowing: Local algorithms for document fingerprinting," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 76–85. [Online]. Available: https://doi.org/10.1145/872757.872770

[60] G. Myers, "A fast bit-vector algorithm for approximate string matching based on dynamic programming," *J. ACM*, vol. 46, no. 3, p. 395–415, may 1999. [Online]. Available: https://doi.org/10.1145/316542.316550

[61] "npm levenshtein distance package." [Online]. Available: https://www.npmjs.com/package/levenshtein

[62] D. Schuler and A. Zeller, "Assessing oracle quality with checked coverage," in *Fourth IEEE International Conference on Software Testing, Verification and Validation*, 2011, pp. 90–99.

[63] P. McMinn, "Search-based software testing: Past, present and future," in *Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Berlin, Germany, 21-25 March, 2011, Workshop Proceedings*. IEEE Computer Society, 2011, pp. 153–163. [Online]. Available: https://doi.org/10.1109/ICSTW.2011.100

[64] A. Panichella, F. M. Kifetew, and P. Tonella, "Automated test case generation as a many-objective optimisation problem with

dynamic selection of the targets," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 122–158, 2018.

[65] M. Pradel and S. Chandra, "Neural software analysis," *Commun. ACM*, vol. 65, no. 1, pp. 86–96, 2022. [Online]. Available: https://doi.org/10.1145/3460348

[66] H. Yu, Y. Lou, K. Sun, D. Ran, T. Xie, D. Hao, Y. Li, G. Li, and Q. Wang, "Automated Assertion Generation via Information Retrieval and Its Integration with Deep Learning," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 163–174. [Online]. Available: https://doi.org/10.1145/3510003.3510149

[67] E. Dinella, G. Ryan, T. Mytkowicz, and S. K. Lahiri, "Toga: A neural method for test oracle generation," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 2130–2141. [Online]. Available: https://doi.org/10.1145/3510003.3510141

[68] Z. Liu, K. Liu, X. Xia, and X. Yang, "Towards more realistic evaluation for neural test oracle generation," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '23, 2023.

[69] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: a database of existing faults to enable controlled testing studies for java programs," in *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, C. S. Pasareanu and D. Marinov, Eds. ACM, 2014, pp. 437–440. [Online]. Available: https://doi.org/10.1145/2610384.2628055

[70] D. M. Stallenberg, M. Olsthoorn, and A. Panichella, "Guess what: Test case generation for Javascript with unsupervised probabilistic type inference," in *Search-Based Software Engineering - 14th International Symposium, SSBSE 2022, Singapore, November 17-18, 2022, Proceedings*, ser. Lecture Notes in Computer Science, M. Papadakis and S. R. Vergilio, Eds., vol. 13711. Springer, 2022, pp. 67–82. [Online]. Available: https://doi.org/10.1007/978-3-031-21251-2_5

[71] K. El Haji, "Empirical study on test generation using GitHub Copilot," Master's thesis, Delft University of Technology, 2023.

[72] E. Andreasen, L. Gong, A. Møller, M. Pradel, M. Selakovic, K. Sen, and C. Staicu, "A survey of dynamic analysis and test generation for JavaScript," *ACM Comput. Surv.*, vol. 50, no. 5, pp. 66:1–66:36, 2017. [Online]. Available: https://doi.org/10.1145/3106739

[73] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A symbolic execution framework for JavaScript," in *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. IEEE Computer Society, 2010, pp. 513–528. [Online]. Available: https://doi.org/10.1109/SP.2010.38

[74] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip, "A framework for automated testing of JavaScript web applications," in *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, R. N. Taylor, H. C. Gall, and N. Medvidovic, Eds. ACM, 2011, pp. 571–580. [Online]. Available: https://doi.org/10.1145/1985793.1985871

[75] G. Li, E. Andreasen, and I. Ghosh, "Symjs: automatic symbolic testing of JavaScript web applications," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, S. Cheung, A. Orso, and M. D. Storey, Eds. ACM, 2014, pp. 449–459. [Online]. Available: https://doi.org/10.1145/2635868.2635913

[76] H. Tanida, T. Uehara, G. Li, and I. Ghosh, *International Journal on Advances in Software*, vol. 8, no. 1/2, 2015.

[77] A. M. Fard, A. Mesbah, and E. Wohlstadter, "Generating fixtures for JavaScript unit testing," in *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, M. B. Cohen, L. Grunske, and M. Whalen, Eds. IEEE Computer Society, 2015, pp. 190–200. [Online]. Available: https://doi.org/10.1109/ASE.2015.26

[78] A. Marchetto and P. Tonella, "Using search-based algorithms for Ajax event sequence generation during testing," *Empir. Softw. Eng.*, vol. 16, no. 1, pp. 103–140, 2011. [Online]. Available: https://doi.org/10.1007/s10664-010-9149-1

[79] S. Mirshokraie and A. Mesbah, "JSART: JavaScript assertion-based regression testing," in *Web Engineering - 12th International Conference, ICWE 2012, Berlin, Germany, July 23-27, 2012. Proceedings*, ser. Lecture Notes in Computer Science, M. Brambilla, T. Tokuda, and R. Tolksdorf, Eds., vol. 7387. Springer, 2012, pp. 238–252. [Online]. Available: https://doi.org/10.1007/978-3-642-31753-8_18

[80] "The Daikon invariant detector," http://plse.cs.washington.edu/daikon/, 2023.

[81] S. Mirshokraie, A. Mesbah, and K. Pattabiraman, "PYTHIA: generating test cases with oracles for JavaScript applications," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, E. Denney, T. Bultan, and A. Zeller, Eds. IEEE, 2013, pp. 610–615. [Online]. Available: https://doi.org/10.1109/ASE.2013.6693121

[82] ——, "JSEFT: automated Javascript unit test generation," in *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*. IEEE Computer Society, 2015, pp. 1–10. [Online]. Available: https://doi.org/10.1109/ICST.2015.7102595

[83] A. M. Fard, M. MirzaAghaei, and A. Mesbah, "Leveraging existing tests in automated test generation for web applications," in *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, I. Crnkovic, M. Chechik, and P. Grünbacher, Eds. ACM, 2014, pp. 67–78. [Online]. Available: https://doi.org/10.1145/2642937.2642991

Max Schäfer is a Principal Software Engineer at GitHub, where he works on program analysis and testing. Previously, he was a Lead Software Engineer at Semmle where he worked on static analysis for Go and JavaScript in CodeQL. He received his PhD in 2010 from the University of Oxford.

Sarah Nadi is an Associate Professor in the Department of Computing Science at the University of Alberta, and a Tier II Canada Research Chair in Software Reuse. She obtained her PhD degree (2014) from the University of Waterloo in Canada and was a post-doctoral researcher (2014-2016) at the Technische Universität Darmstadt in Germany. Her research interests include mining software repositories and empirical software engineering, especially w.r.t the selection, usage, and migration of software libraries.

Aryaz Eghbali is a PhD student at the University of Stuttgart, advised by Dr. Michael Pradel. He received his MSc from the University of Calgary in 2018. His research interests include dynamic program analysis and AI-based developer tools.

Frank Tip received the PhD degree from the University of Amsterdam, Amsterdam, Netherlands, in 1995. He is currently a professor at the Khoury College of Computer Sciences, Northeastern University, Boston, Massachusetts. His research interests include static and dynamic program analysis, asynchronous programming, refactoring, and test generation.