Unify and Triumph: Polyglot, Diverse, and Self-Consistent Generation of Unit Tests with LLMs

Djamel Eddine Khelladi
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
djamel-eddine.khelladi@irisa.fr

Charly Reux
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
charly.reux@inria.fr

Mathieu Acher
Univ Rennes, Inria, CNRS, IUF, IRISA
Rennes, France
mathieu.acher@irisa.fr

ABSTRACT

Large language model (LLM)-based test generation has gained attention in software engineering, yet most studies evaluate LLMs' ability to generate unit tests in a single attempt for a given language, missing the opportunity to leverage LLM diversity for more robust testing. This paper introduces *PolyTest* a novel approach that enhances test generation by exploiting polyglot and temperature-controlled diversity. *PolyTest* systematically leverages these properties in two complementary ways: (1) Cross-lingual test generation: Tests are generated in multiple languages at zero temperature and then unified; (2) Diverse test sampling: Multiple test sets are generated within the same language at a higher temperature before unification. A key idea is that LLMs can generate diverse yet contradicting tests – same input, different expected outputs – across languages and generations. *PolyTest* mitigates these inconsistencies by unifying test sets across languages and generations, fostering self-consistency and improving overall test quality.

We evaluate *PolyTest* on *LLama3-70b*, *GPT-4o*, and *GPT-3.5* using the latest version of the dataset EvalPlus that contains curated prompts of coding problems and canonical solutions. On 164 problems, we generate tests at temperature 0 for five languages, namely Java, C, Python, JavaScript, and also in a language-agnostic format of input/output in a CSV. We also generate tests five times for each of the above languages with a high temperature at 1. We perform the union of the tests at each step with our *PolyTest* approach (i.e., between the 5 languages and between the 5 generations per language). We observe up to **15.42%** contradicting tests (in JavaScript with GPT-4o), **7.41%** (in C with Llama3-70b), and **6.51%** (in Java with GPT-3.5). Results also show that *PolyTest* in both polyglot and temperature-controlled diversity is indeed able to improve the obtained tests w.r.t. all metrics we considered, namely number of tests and of passing tests (multiplied up to **x2.67** and **x2.85**), statement and branch coverage (up to **+7.9%** and **+9.01%**), and mutation score (up to **+11.23%**). Overall, *PolyTest* outperforms single-language and single-attempt approaches without requiring on-the-fly execution of every test case, and is particularly beneficial for programming languages where LLMs exhibiting weak performance. Finally, *PolyTest* also outperformed Pynguin, as a baseline comparison, in generated/passing tests and mutation score.

KEYWORDS

LLM, LLama, GPT, Multi-lingual, Polyglot, Temperature, Tests.

1 INTRODUCTION

Large language models (LLMs) have emerged in the field of natural language processing, exhibiting high aptitude to transform and generate textual data. Since their appearance, LLMs have been applied in different domains and tasks in Software Engineering [2, 3, 9, 10, 15, 18, 20, 21, 23, 28, 32–34, 37, 41, 42, 46, 48].

Testing is a crucial part of software development to ensure quality and correctness of software. However, manually specifying and writing relevant tests is a non-trivial task. Hence, an extensive literature emerged on automatic unit test[1] generation among which lately LLM-based test generation has attracted attention [43]. In fact, LLMs stand as promising tools for tackling increasingly complex problems and support developers in various tasks of writing, correcting and documenting source code and other artifacts. While there is an extensive empirical assessment of the LLMs capabilities in generating code [2, 3, 9, 10, 15, 18, 20, 21, 23, 28, 32–34, 37, 41, 42, 46, 48], there are less works assessing their ability to generate tests [8, 26, 38–40]. However, to the best of our knowledge, they only evaluate the capability of LLMs to generate better tests in one shot for a target single language or set of languages, or improve the tests with static analysis, mutation, or repair [14, 19, 35].

This paper introduces *PolyTest* a novel approach that enhances tests generation by exploiting the diversity of LLMs' output induced by multi-lingual (a.k.a. polyglot) and temperature-control. In fact, one of the powerful diversity features of LLMs is their polyglot nature, i.e., trained on multiple languages, and hence, capable of handling tasks across multiple languages. For example, LLMs can translate code from a source language to a different target language, and generating code and tests for multiple languages, such as Java, C, Python, etc. LLMs can also produce diverse outputs because the temperature parameter controls the balance between creativity and predictability when sampling. *PolyTest*'s novel idea is to systematically leverage these properties in two complementary ways: *(1) Cross-lingual test generation*: Tests are generated in *n* multiple languages at temperature zero and then unified; *(2) Diverse test sampling*: *n* Multiple test sets are generated within the same language at a higher temperature before unification. A key idea is that LLMs may produce tests with the same input but conflicting expected outputs across languages and generations – a problematic inconsistency. *PolyTest* addresses the challenge of contradictory tests by unifying outputs generated across different languages and multiple generations. Rather than depending on a single test output, *PolyTest* samples multiple candidate tests and reconciles them—ensuring self-consistency, resolving contradictions, and uncovering potential cases that a one-shot LLM generation might miss. Notably, this unification and contradiction detection process does not require on-the-fly execution of every test case. We further consider an additional step to amplify them, which would foster the diversity of the tests and ultimately to enhance their quality.

---

[1]For simplicity, we will refer to tests rather than unit tests in the rest of the paper.

```
(User prompt)
Write the test cases for the following function def derivative(xs: list):
""" xs represent coefficients of a polynomial.
xs\[0\] + xs\[1\] \* x + xs\[2\] \* x^2 + ....
Return derivative of this polynomial in the same form.
"""
```

Figure 1: A prompt for the derivative of a polynomial.

We evaluate our implementation of *PolyTest* on three LLMs, namely *LLama3-70b*, *GPT-4o*, and *GPT-3.5*. We use the latest version of the popular dataset of EvalPlus[2] [29] that contains curated prompts of coding problems and canonical solutions (i.e., reference code). We evaluate our approach on 164 problems at a temperature = 0. For each problem, we generate and amplify tests in four programming languages (Java, C, Python, and JavaScript) and in a language-agnostic input/output format (CSV). We then, performed the same five times per individual language at temperature = 1, hence, covering both setups of *PolyTest*. We perform the union of the tests at each step with our *PolyTest* approach, i.e., between the 5 languages and between the 5 generations per language. We observe up to **15.42%** contradicting tests (in JavaScript with GPT-4o), **7.41%** (in C with Llama3-70b), and **6.51%** (in Java with GPT-3.5). Results also show that *PolyTest* is indeed able to improve the obtained tests w.r.t. all metrics we considered, namely number of tests and passing tests (multiplied up to **x2.67** and **x2.85**), statement and branch coverage (improved up to **+7.9%** and **+9.01%**), and mutation score (improved up to **+11.23%**). Ultimately, *PolyTest* outperforms single-language and single-attempt approaches without requiring continuous executions of test cases. *PolyTest* can be particularly beneficial for programming languages where LLMs exhibiting weak performance and low test quality. Finally, *PolyTest* also outperformed Pynguin, as a baseline comparison, in generated/passing tests and mutation score, with an equivalent coverage.

To summarize, our main contributions are as follow:

(1) A Novel approach *PolyTest* to enhance tests generation with. To the best of our knowledge, it is the first automatic approach taking advantage of the diversity induced by the polyglot feature and temperature-control of LLMs.

(2) We report a qualitative analysis of how equivalent, different, and contradicting obtained tests are between different languages and between multiple generations per language.

(3) Empirical evaluation of *PolyTest* and comparison with five languages, five generations, and to Pynguin as a baseline, showing gains and best performance with *PolyTest* for both steps of generation and amplification.

(4) Publicly available implementation and results on the EvalPlus benchmark [1] for reproducibility.

2 MOTIVATING EXAMPLE

This section introduces a motivating example to illustrate the test generation for multiple languages and the effect of their unification.

Let us take as an example a prompt specification for the problem of computing the derivative of a polynomial, as shown in Figure 1. Figure 2 shows the results of test generation at temperature zero for four target languages, namely for Java, Javascript (JS), C, and

```
(LLM test generation for Java)
1. assertEquals(new int[]{1, 4, 12, 20},
derivative(new int[]{3, 1, 2, 4, 5}));
2. assertEquals(new int[]{2, 6}, derivative(new int[]{1, 2, 3}));
3. assertEquals(new int[]{2}, derivative(new int[]{1, 2}));
4. assertEquals(new int[]{}, derivative(new int[]{5}));
5. assertEquals(new int[]{}, derivative(new int[]{}));
```

```
(LLM test amplification for Java)
6. assertEquals(new int[]{0}, derivative(new int[]{0}));
7. assertEquals(new int[]{1}, derivative(new int[]{0, 1}));
8. assertEquals(new int[]{2, 4}, derivative(new int[]{0, 0, 2}));
```

```
(LLM test generation for C)
1. assert(derivative([3, 1, 2, 4, 5]) == [1, 4, 12, 20]);
2. assert(derivative([1, 2, 3]) == [2, 6]);
3. assert(derivative([5]) == []);
4. assert(derivative([1, 2]) == [2]);
5. assert(derivative([1]) == []);
```

```
(LLM test amplification for C)
6. assert(derivative([5, 0, 0, 0]) == [0, 0, 0]);
7. assert(derivative([0, 0, 0, 0]) == [0, 0, 0, 0]);
8. assert(derivative([1, 1, 1, 1]) == [1, 2, 3, 4]);
9. assert(derivative([10]) == []);
10. assert(derivative([]) == []);
```

```
(LLM test generation for JS)
1. assert.deepEqual(derivative([3, 1, 2, 4, 5]), [1, 4, 12, 20]);
2. assert.deepEqual(derivative([1, 2, 3]), [2, 6]);
3. assert.deepEqual(derivative([5]), []);
4. assert.deepEqual(derivative([1, 2]), [2]);
5. assert.deepEqual(derivative([1]), []);
```

```
(LLM test amplification for JS)
6. assert.deepEqual(derivative([1, 1, 1, 1]), [1, 2, 3]);
7. assert.deepEqual(derivative([]), []);
8. assert.deepEqual(derivative([0, 0, 0]), [0, 0]);
```

```
(LLM test generation for Python)
1. assert derivative([3, 1, 2, 4, 5]) == [1, 4, 12, 20]
2. assert derivative([1, 2, 3]) == [2, 6]
3. assert derivative([5, 0, 0, 0]) == [0, 0, 0]
4. assert derivative([0, 0, 0, 0]) == [0, 0, 0]
5. assert derivative([1]) == []
6. assert derivative([]) == []
```
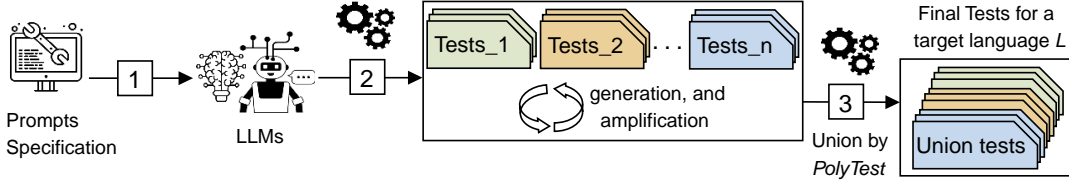
```
(LLM test amplification for Python)
6. assert derivative([1, 1, 1, 1]) == [1, 2, 3, 4]
7. assert derivative([5]) == []
```

Figure 2: LLM generated and amplified tests for Java, JS, C, and Python.

Python. They are in the format of an *assert* with an *input* for the derivative function and an expected *output*.

We first observe that the LLM generates similar tests when prompted with the same problem to solve in multiple languages. However, it does generate different tests in some cases. For example, the last test $n^o 5$ in Java is not present in JS and C, and vice versa,

Figure 3: Overall approach of *PolyTest*. It covers two setups: 1) One generation of tests for *n* languages and 2) *n* generations for a single language. It also include three steps, generation, amplification, and reduction of tests.

while both tests are present in Python. In addition, tests n^o3 and n^o4 in Python are not generated in other languages. This gives us a solid hint that indeed LLMs may generate different tests depending on the target language. Moreover, when asking the LLM to amplify the tests, we start to observe real divergences, i.e., tests with different inputs and outputs. For example, the test n^o8 in Java is not proposed in any of the other four languages. Similarly for the test n^o6 in JS and the tests n^o9 in C. As generated tests can differ for the same prompt problem on different languages, it is a diversity to take advantage of to explore unification for enhancing the test suite and various quality metrics, such as coverage and mutation score. Furthermore, this is likely also true when multiple generations for the same language is performed with a high temperature to allow for more creativity and diversity. For example, generating four times at temperature 1 in Java rather than one time in Java, JS, C, and Python. Ultimately, to unify the generated tests and enhance the quality of the test suite.

However, to the best of our knowledge there is no approach allowing to automatically leverage this diversity of generated tests based on the LLM polyglot and temperature, and more importantly no empirical evaluation exists on how much unifying LLM's generated tests improves the test suite. We fill this gap in this work.

3 APPROACH

This section introduces our approach *PolyTest*. The rationale and vision behind it is to reach a consensus through self-consistency for test suite and its quality. A given LLM can be weak in testing one language or in one shot iteration, but strong in testing another language or another iteration. This strength induced by the LLMs diversity can be unified to be capitalized on to improve the tests. This can be seen as a self-consistent approach.

Figure 3 shows the overall approach and its workflow. The first step ☐1☐ is to use a given LLM to then generate tests for a set of prompts ☐2☐. Here two setups are covered, namely: 1) one generation of tests for *n* different languages and 2) *n* generations for a single language. After that, *PolyTest* performs a union of the different sets of tests ☐3☐ by aggregating the different tests and translating them in one chosen target language. In this step, *PolyTest* removes duplicates and only keeps a unique occurrence of each test.

One particularity of *PolyTest* is its treatment of the tests in two distinct steps. First, it asks a given LLM to generate tests from the prompts. Then, *PolyTest* asks the LLM to amplify the tests, i.e., to generate even more relevant tests. Herein, the amplification is done on top of the generated ones. The rationale behind the amplification step is to let the LLM propose other tests likely covering different inputs and scenarios. *PolyTest* collects the different sets of tests at each step. Hence, we can compare their quality later on. However,

---

Algorithm 1: *PolyTest* in first scenario of one generation in multiple languages.

Data: Prompt, LLM, Target Laguage TL, Languages
1 p ← Prompt
2 test_res ← Languages.size() * 3 Matrix
3 i ← 0
4 **for** *(l ∈ Languages)* **do**
5   tests_step_1 ← generationRequest(LLM, l, p) */*prompting the LLM to generate tests*/*
6   tests_step_2 ← amplificationRequest(LLM, l, p, tests_step_1) */*prompting the LLM to amplify the generated tests*/*
7   test_res[i, 0].add(tests_step_1) */*storing the tests of the two steps per language*/*
8   test_res[i, 1].add(tests_step_2)
9   i++
10 **end**
11 unionTests ← UnionTests(test_res, LLM, TL) */*Algorithm 3*/*

---

the developers using *PolyTest* can freely use the results of one of the two steps w.r.t. their needs.

Algorithms 1 and 2 details how *PolyTest* works for the two setups, respectively, one generation of tests for *n* different languages and *n* generations for a single language. In Algorithm 1, given a prompt *p* specifying a given problem, an *LLM*, a target language and a list of languages (lines 0-2), *PolyTest* will generate and amplify the tests for all the *l* different languages (Lines 4-11). It first requests the LLM to generate a set of tests for *p* in each language *l* (Line 5). Then, it will request to amplify the tests (Line 6). It does so by ending the prompt with respectively, *"Generate unit tests."* and *"Amplify the provided unit tests."*. The tests results for each step are stored for each language (Lines 7-8). Similarly, in Algorithm 2, given a prompt *p* specifying a given problem, an *LLM*, a target language and a number of generations *NbrGen* (lines 0-2), *PolyTest* will generate and amplify the tests *NbrGen* times for the target language *TL* (Lines 4-11). After that, for each of the two steps, *PolyTest* perform in Algorithm 3 the union of the multiple sets of tests (Lines 3-8). It then converts the tests from the different languages into one chosen target language *TL*. The conversation is done through the LLM (Lines 5-7). Tests are added once in the final unified set of tests, hence, ignoring duplicates. Ultimately, *PolyTest* resulting also in unified tests corresponding to the two steps of generation and amplification.

4 METHODOLOGY

This section describes our empirical evaluation of *PolyTest* and whether leveraging the diversity of LLMs through polyglot feature and temperature change would yield better results for test generation. The section first presents the selected LLMs, then our dataset, research questions, and finally the evaluation process.

Algorithm 2: *PolyTest* in second scenario of multiple generations in one single language.

Data: Prompt, LLM, Target Laguage TL, number of generation NbrGen

1 p ← Prompt
2 test_res ← NbrGen * 3 Matrix
3 **for** *(i ∈ NbrGen)* **do**
4   tests_step_1 ← generationRequest(LLM, TL, p) */*prompting the LLM to generate tests*/*
5   tests_step_2 ← amplificationRequest(LLM, TL, p, tests_step_1) */*prompting the LLM to amplify the generated tests*/*
6   test_res[i, 0].add(tests_step_1) */*storing the tests of the two steps per generation*/*
7   test_res[i, 1].add(tests_step_2)
8   i++
9 **end**
10 unionTests ← UnionTests(test_res, LLM, TL) */*Algorithm 3*/*

Algorithm 3: UnionTests()
Union of generated, amplified, and reduced tests.

Data: tests_matrix, LLM, Target Laguage TL
1 tests ← tests_matrix
2 unionTests ← {ϕ} */*a set with non-duplicate elements*/*
3 **for** *(t ∈ tests)* **do**
4   */*unify the tests for each of the generated and amplified tests for all languages or generations*/*
5   unionTests[0] ← unionTests[0] ∪ convertTest(LLM, TL, t[0])
6   unionTests[1] ← unionTests[1] ∪ convertTest(LLM, TL, t[1])
7 **end**
8 **return** unionTests

4.1 Selected LLMs and parameterization

We chose *llama3-70b*, *GPT-4o*, and *GPT-3.5*. These three models are popular with with good performances that are accessible for us and available with no or a small cost. Thus, easing future replication and reproduction of our results. The *temperature hyperparameter* is usually suggested to be set between 0 and 1 in the documentation. The lower the temperature, the more deterministic the results are (e.g., at 0). Increasing temperature could lead to more diversity, creativity, and randomness. Therefore, we set the temperature of the LLMs to different values for the two setups of *PolyTest*. When generating tests once for *n* different languages, we set the temperature to zero (0). Thus, we only leverage the diversity brought by the multiple polyglot languages. When generating *n* times tests for a single language, we set the temperature to one (1), hence, leveraging only on the diversity brought by the high temperature.

4.2 Dataset

This section details our selected dataset. We chose EvalPlus[3] [29], the lastest up-to-date dataset that builds on top of two existing benchmarks, namely HumanEval[4] [13] and MBPP[5] [6]. EvalPlus benchmark is a dataset designed to evaluate the code generation capabilities of large language models (LLMs). It has 164 problems

---

[3] https://github.com/evalplus/evalplus
[4] https://github.com/openai/human-eval
[5] https://github.com/google-research/google-research/tree/master/mbpp

---

consisting of hand-crafted programming problems, each including a function signature, docstring, and body of canonical solution (i.e., reference code) that is important in our evaluation to verify the correctness of the obtained tests.

4.3 Research Questions

This section presents the research questions for our empirical study and evaluation of *PolyTest* for test generation w.r.t. single general purpose languages (GPLs).

RQ1 Does *PolyTest* increases the number of obtained tests? This aims to quantify the improvements or not in terms of number of tests.

RQ2 How much obtained tests are equivalent, different, and in contradiction across the different GPLs? This aims to assess how the obtained tests in different languages or generations are alike, different, and even contradicting.

RQ3 How do obtained tests that pass or fail vary across GPLs and *PolyTest*? This aims to check the correctness of the obtained tests before and after being unified with *PolyTest*. Note that failing tests are supposedly wrong, since we have the canonical solutions to check this in our dataset.

RQ4 How much statement coverage and branch coverage vary across GPLs and *PolyTest*? This aims to check the quality of the obtained tests before and after being unified with *PolyTest* with the coverage metric.

RQ5 How does mutation score vary across GPL and *PolyTest*? This aims to check a crucial quality metric of the obtained tests before and after being unified with *PolyTest*.

RQ6 How does *PolyTest* compare to a baseline of test generation? This positions *PolyTest* with SOTA Pynguin tool [30].

4.4 Evaluation Process

We launched *PolyTest* for each of the prompts in our dataset by using the APIs of our two LLMs. We chose to include in *PolyTest* the following languages: Java, C, Python, JavaScript, and also to ask for language-agnostic tests in the form of input/output in a CSV format. We then chose Python as a target language for unification of tests, but it could have been any other language. In fact the target language does not change the results of *PolyTest*, since the unification algorithms will not be impacted and remains the same. Note that the tests for the other languages are translated with the LLM to Python to execute them on the canonical solutions. We later on checked a random subset and confirmed the correctness of the translation (see section 5.8). We store all intermediate and final results, i.e., tests per language, unified tests, for generation and amplification steps. Thus, we could later on compare them, check their correctness and quality by computing various quality metrics. In particular, *statement + branch coverage* and *mutation score*. We reuse coverage.py[6] and mut.py[7] tools to compute the coverage and mutation metrics. They are computed as follows:

$$statement\ coverage = \frac{nbr\ of\ visited\ Statements}{nbr\ of\ total\ Statements} \times 100$$

$$branch\ coverage = \frac{nbr\ of\ visited\ branches}{nbr\ of\ total\ branches} \times 100$$

---

[6] https://coverage.readthedocs.io/en/7.5.0/
[7] https://github.com/mutpy/mutpy

$$mutation\ score = \frac{nbr\ of\ killed\ mutants}{nbr\ of\ total\ mutants} \times 100$$

Our dataset and implementation are publicly available in [1].

5 RESULTS

As we evaluate on Java, C, Python, JS, and CSV, in the remaining of the results section, we will refer to obtained tests with *PolyTest* from one generation for 5 different languages as $PolyTest_{5\_lang}$ and form 5 generations per language as $PolyTest_{Java\times5}$, $PolyTest_{C\times5}$, $PolyTest_{Python\times5}$, $PolyTest_{JS\times5}$, and $PolyTest_{CSV\times5}$.

5.1 RQ1

First we ran our experimental protocol to obtain the tests to unify with *PolyTest* in the different setups.

Column 4 in Table 1 for *llama3-70b*, Table 2 for *GPT-4o*, and Table 3 for *GPT-3.5* gives the total number of obtained test for the generation and amplification steps of our approach. We observe that *PolyTest* outperforms other single language generations in number of total tests. For *llama3-70b*, *PolyTest* (in all six setups) multiples, on average, the generated and amplified tests by respectively x2.16 and x2.67. For *GPT-4o*, *PolyTest* multiples, on average, the generated and amplified tests by respectively x2.16 and x2.67. For *GPT-3.5*, *PolyTest* multiples, on average, the generated and amplified tests by respectively x1.8 and x2.3. *PolyTest* thus increases significantly the number of tests compared to each single language.

> **RQ1 insights:** All *PolyTest* setups allows to increase the number of generated and amplified tests, respectively, by **x2.16** and **x2.67** for *llama3-70b*, by **x2.16** and **x2.67** for *GPT-4o*, and by **x1.8** and **x2.3** for *GPT-3.5*. This is non-negligible gains.

5.2 RQ2

To answer this question, we compared the obtained tests between the different languages for $PolyTest_{5\_lang}$ and between the five generations per language for $PolyTest_{Java\times5}$, $PolyTest_{C\times5}$, $PolyTest_{Python\times5}$, $PolyTest_{JS\times5}$, and $PolyTest_{CSV\times5}$. In particular, we compared them pairwise two by two.

First, we observe that great number of tests are equivalent and do overlap between the different pairs of languages and generations per language. The overlap is observed more with $PolyTest_{5\_lang}$ than with the other setups. Due to lack of space the figures illustrating the overlap percentages (%) are in our data set accessible online [1]. For *llama3-70b*, the overlap varies from 3.62% up to 78.7% in the generated tests and from 4.71% up to 62.5% in the amplified tests. For *GPT-4o*, the overlap varies from 4.11% up to 80.5% in the generated tests and from 5.17% up to 53% in the amplified tests. For *GPT-3.5*, the overlap is more present in the generation step compared to the amplification step, especially with $PolyTest_{5\_lang}$. The overlap varies from 1% up to 94.5% in the generated tests and from 1% up to 64.2% in the amplified tests. Still, an important part of the generated and amplified tests overall differed between the different languages and generations.

We further looked at the contradictions between the tests in different languages and generations. Our hypothesis is that generated tests for the same prompt problem are not contradicting themselves. Meaning that if they have the same input, they should have the same
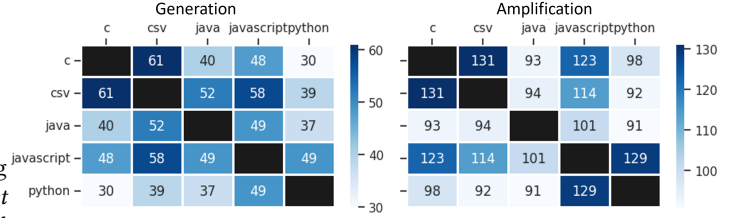

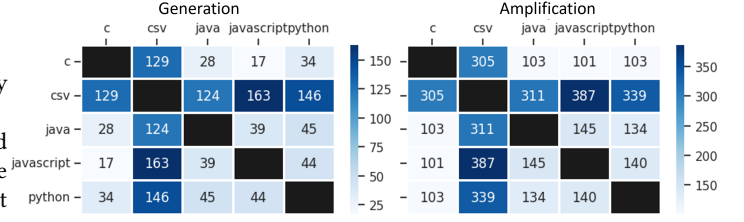**Figure 4: Contradiction amid language pairs in *llama3-70b*.**

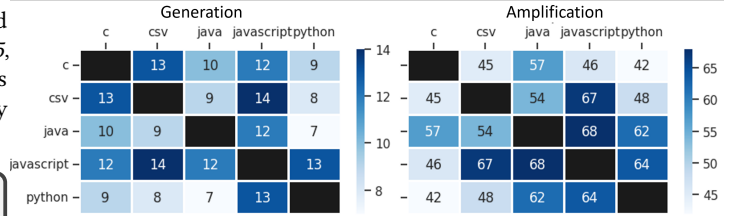**Figure 5: Contradiction amid language pairs in *GPT-4o*.**

Figure 6: Contradiction amid language pairs in *GPT-3.5*.

output as well. To verify our hypothesis, we searched for tests sharing the same input but have different outputs. Figures 4, 5, 6 show the number of contradicting tests between each pair of languages and Figures 7, 8 show the number of contradicting tests between the different generations per language for *llama3-70b* and *GPT-4o* (we ommit the Figure for *GPT-3.5* due to lack of space). We first observe that the three LLMs do generate contradicting tests in both generation and amplification steps, which rejects our hypothesis. We observe slightly more contradicting tests with $PolyTest_{5\_lang}$ than with $PolyTest_{Java\times5}$, $PolyTest_{C\times5}$, $PolyTest_{Python\times5}$, $PolyTest_{JS\times5}$, and $PolyTest_{CSV\times5}$.

While these cases still represent only a small part of all the obtained tests, we nonetheless observe them between almost all pairs of languages or generations. In particular, with a maximum of contradicting tests in the amplified step up to 15.42% in JavaScript with *GPT-4o*, 7.41% in C with *Llama3-70b*, and 6.51% in Java with *GPT-3.5*. Overall, the maximums of contradicting tests in the different *PolyTest* setups were as follows. For *llama3-70b*, 131 in C with $PolyTest_{5\_lang}$, 78 in C with $PolyTest_{C\times5}$, 141 in CSV with $PolyTest_{CSV\times5}$, 72 in Java with $PolyTest_{Java\times5}$, 17 in Javascript with $PolyTest_{JS\times5}$, and 58 in Python with $PolyTest_{Python\times5}$. For *GPT-4o*, the maximum of contradicting tests was 387 in Javascript with $PolyTest_{5\_lang}$, 46 in C with $PolyTest_{C\times5}$, 152 in CSV with $PolyTest_{CSV\times5}$, 67 in Java with $PolyTest_{Java\times5}$, 12 in Javascript with $PolyTest_{JS\times5}$, and 58 in Python with $PolyTest_{Python\times5}$. For *GPT-3.5*, the maximum of contradicting tests was 68 in Java with $PolyTest_{5\_lang}$, 36 in C with $PolyTest_{C\times5}$, 25 in CSV with

Djamel Eddine Khelladi, Charly Reux, and Mathieu Acher

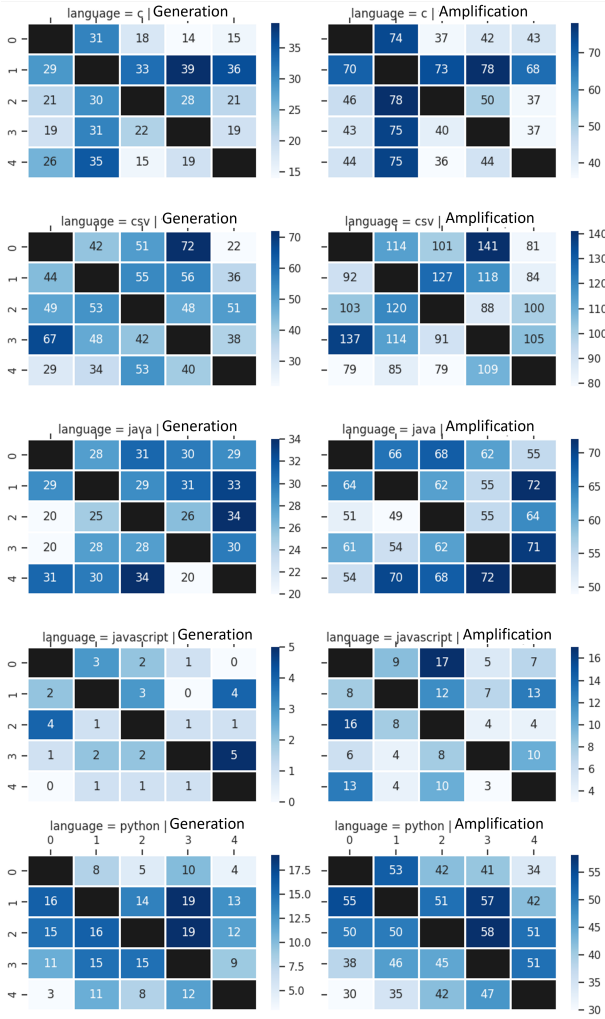Figure 7: Contradiction amid pairs in the five generations per language for *llama3-70b*.

Figure 8: Contradiction amid pairs in the five generations per language for *GPT-4o*.

$PolyTest_{CSV\times5}$, 21 in Java with $PolyTest_{Java\times5}$, 3 in Javascript with $PolyTest_{JS\times5}$, and 27 in Python with $PolyTest_{Python\times5}$.

These cases of contradicting tests emphasize the need to verify and validate the correctness of the tests. *PolyTest* can be seen as a kind of self-consistency validation since it will consider the different contradicting tests between the different languages. Hence, *PolyTest* will likely keep the correct passing ones at the end, which could not be the case for a single language or a single generation. In our case, since we have the canonical solutions, we can filter the wrong contradictions that will make the tests fail from those that pass. We further look into this in the next RQ.
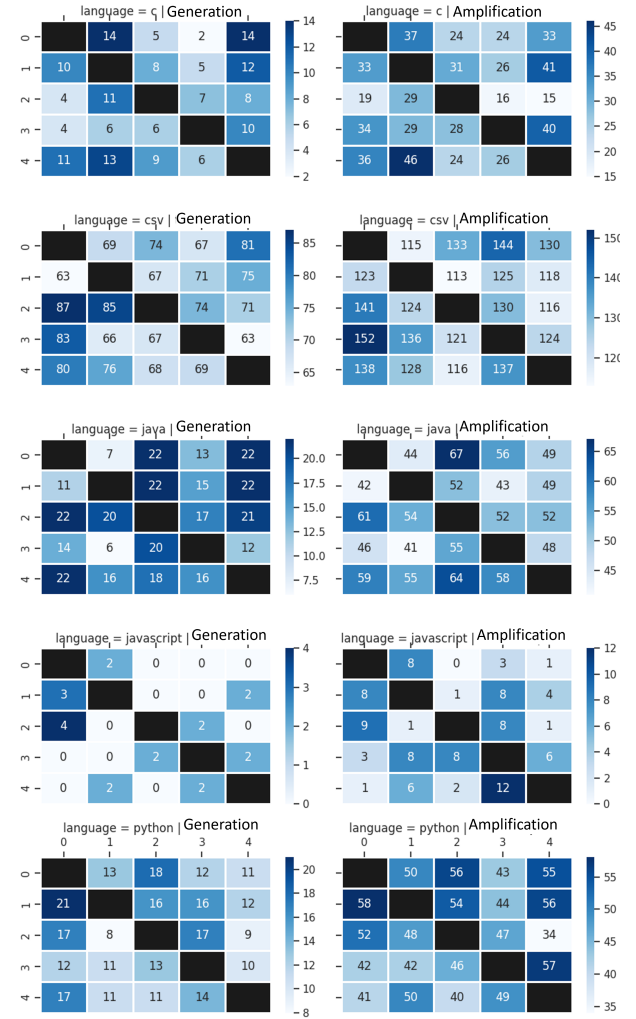
> **RQ_2 insights:** A great number of tests are in common between the different languages and generations, yet with a flagrant diversity. Surprisingly, contradicting tests were systematically observed in between almost all pairs of languages and the multiple generations per language in all three LLMs. The maximum of contradicting tests were up to **15.42%** with *GPT-4o*, **7.41%** with *Llama3-70b*, and **6.51%** with *GPT-3.5*.

5.3 RQ3

To answer this question, we rely on the canonical solutions in our dataset that are known to be the correct implementation solution. Thus, if a test passes, we consider it correct and if it fails, we consider it incorrect. This way, we can check the correctness of the obtained tests for each language and after being unified with *PolyTest*.

Column 5 in Tables 1, 2, 3 shows the number of passing tests per language and for the union with *PolyTest* for the generated and amplified tests in the the LLMs. We observe that *PolyTest* outperforms

Table 1: Results for *PolyTest* with llama3-70b.

Temperature	Language	Step	n^o of total test	n^o of passing tests	Statement coverage	Branch coverage	Mutation score
temp=0	C	Gen.	983	814	94.71%	93.40%	83.73%
		Ampl.	1769	1291	92.18%	90.42%	82.74%
	CSV	Gen.	1112	912	96.16%	95.19%	87.66%
		Ampl.	2336	1774	96.45%	95.42%	88.63%
	Java	Gen.	843	737	98.44%	97.32%	88.70%
		Ampl.	1487	1181	98.23%	97.29%	88.34%
	Javascript	Gen.	973	831	97.68%	96.65%	87.93%
		Ampl.	1745	1379	97.55%	96.50%	88.46%
	Python	Gen.	994	858	98.90%	98.09%	89.21%
		Ampl.	1843	1455	98.85%	97.94%	90.09%
	$PolyTest_{5\_lang}$	Gen.	2180	1634	99.05%	98.34%	91.71%
		Ampl.	5179	3543	98.94%	97.87%	93.53%
temp=1	$PolyTest_{C\times5}$	Gen.	2207	1496	98.49%	97.71%	88.92%
		Ampl.	4714	2999	98.91%	98.23%	91.40%
	$PolyTest_{CSV\times5}$	Gen.	2636	1884	97.97%	97.38%	90.48%
		Ampl.	5999	3814	97.69%	97.03%	90.43%
	$PolyTest_{Java\times5}$	Gen.	1833	1337	99.06%	98.43%	90.85%
		Ampl.	4127	2722	99.38%	98.83%	92.08%
	$PolyTest_{JS\times5}$	Gen.	2092	1576	99.28%	98.55%	91.51%
		Ampl.	4652	3180	99.46%	98.85%	92.52%
	$PolyTest_{Python\times5}$	Gen.	2058	1623	99.39%	98.84%	92.17%
		Ampl.	4848	3405	99.39%	98.92%	93.31%

Table 2: Results for *PolyTest* with GPT-4o.

Temperature	Language	Step	n^o of total test	n^o of passing tests	Statement coverage	Branch coverage	Mutation score
temp=0	C	Gen.	981	891	97.93%	97.01%	89.82%
		Ampl.	2680	2281	98.48%	97.56%	91.02%
	CSV	Gen.	1671	1283	91.63%	90.15%	83.12%
		Ampl.	4626	3325	91.71%	90.25%	83.53%
	Java	Gen.	973	849	98.62%	97.63%	89.95%
		Ampl.	2677	2166	98.37%	97.47%	90.83%
	Javascript	Gen.	1157	1029	98.20%	97.27%	90.18%
		Ampl.	3022	2510	96.82%	95.79%	89.22%
	Python	Gen.	1260	1158	99.09%	98.39%	91.83%
		Ampl.	3354	2927	99.04%	98.17%	93.23%
	$PolyTest_{5\_lang}$	Gen.	2903	2249	99.48%	98.98%	93.33%
		Ampl.	9975	7311	99.61%	98.87%	94.39%
temp=1	$PolyTest_{C\times5}$	Gen.	1480	1214	99.03%	98.32%	90.14%
		Ampl.	5170	4108	99.54%	99.04%	94.45%
	$PolyTest_{CSV\times5}$	Gen.	3671	2865	99.60%	99.16%	93.79%
		Ampl.	8937	6625	99.46%	98.79%	94.76%
	$PolyTest_{Java\times5}$	Gen.	2389	1822	99.39%	98.73%	91.96%
		Ampl.	6204	4386	99%	98.34%	92.67%
	$PolyTest_{JS\times5}$	Gen.	2657	2167	99.23%	98.65%	92.12%
		Ampl.	6581	5073	99.04%	98.50%	94.07%
	$PolyTest_{Python\times5}$	Gen.	2890	2420	99.43%	98.93%	93.30%
		Ampl.	7309	5788	99.59%	99.10%	94.64%

Table 3: Results for *PolyTest* with GPT-3.5.

Temperature	Language	Step	n^o of total test	n^o of passing tests	Statement coverage	Branch coverage	Mutation score
temp=0	C	Gen.	497	454	93.86%	91.92%	83.59%
		Ampl.	1037	831	94.54%	92.80%	80.92%
	CSV	Gen.	471	445	96.73%	94.70%	85.17%
		Ampl.	1235	1028	98.32%	96.94%	89.97%
	Java	Gen.	485	457	96.84%	94.84%	85.25%
		Ampl.	1045	802	97.18%	95.53%	86.34%
	Javascript	Gen.	516	477	95.79%	94.05%	83.89%
		Ampl.	1111	884	96.34%	94.75%	86.12%
	Python	Gen.	501	475	96.43%	94.64%	84.93%
		Ampl.	1134	906	97.40%	95.97%	88.78%
	$PolyTest_{5\_lang}$	Gen.	659	540	98.47%	96.92%	87.35%
		Ampl.	2582	1688	98.67%	97.50%	91.57%
temp=1	$PolyTest_{C\times5}$	Gen.	598	527	96.99%	95.16%	84.49%
		Ampl.	1637	945	95.22%	93.11%	85.60%
	$PolyTest_{CSV\times5}$	Gen.	543	471	98.22%	96.41%	87.55%
		Ampl.	2745	1761	98.41%	96.39%	85.99%
	$PolyTest_{Java\times5}$	Gen.	594	504	98.65%	97.22%	88.21%
		Ampl.	2276	1214	98.12%	96.26%	84.38%
	$PolyTest_{JS\times5}$	Gen.	658	565	98.99%	97.52%	88.73%
		Ampl.	2294	1452	98.52%	96.74%	89.03%
	$PolyTest_{Python\times5}$	Gen.	706	612	98.52%	97.12%	88.55%
		Ampl.	2729	1756	98.36%	96.74%	89.95%

other single languages in number of total correct tests passing that will be kept by developers at the end. For *llama3-70b*, *PolyTest* (in all six setups) multiples, on average, the generated and amplified passing tests by respectively x1.92 and x2.35. For *GPT-4o*, *PolyTest* multiples, on average, the generated and amplified passing tests by respectively x2.62 and x2.85. For *GPT-3.5*, *PolyTest* multiples, on average, the generated and amplified passing tests by respectively x1.16 and x1.66. *PolyTest* thus increases significantly the number of passing tests compared to each single language, especially with *llama3-70b* and *GPT-4o*. While *PolyTest* increases the passing tests compared to each single language, it does not mean that the overall quality is improved. The next RQs investigate this aspect.

> **RQ3 insights:** *PolyTest* increased the passing tests. With *llama3-70b* and *GPT-4o*, *PolyTest* more than double the passing tests, on average, up to **x2.35** and **x2.85**. For *GPT-3.5*, *PolyTest* multiples the passing tests, on average, by **x1.66**.

5.4 RQ4

To answer this question, we compute statement and branch coverage for the obtained passing tests.

Column 6 and 7 in Tables 1, 2, 3 give the average statement and branch coverage per language and all *PolyTest* setups for the generated and amplified tests. Once again, for all three LLMs, we observe that the highest coverage metrics are obtained by *PolyTest* (in all its setups) exceeding all other languages in both steps. For *llama3-70b*, we observe gains in: *(1)* statement coverage up to 4.68% in the generated tests and up to 7.21% in the amplified tests, *(2)* branch coverage

up to 5.44% in the generated tests and up to 8.5% in the amplified tests. For *GPT-4o*, we observe gains in *(1)* statement coverage up to 7.85% in the generated tests and up to 7.9% in the amplified tests, *(2)* branch coverage up to 9.01% in the generated tests and up to 8.85% in the amplified tests. Finally, for *GPT-3.5-turbo*, we observe gains in *(1)* statement coverage up to 5.13% in the generated tests and up to 4.13% in the amplified tests, *(2)* branch coverage up to 5.6% in the generated tests and up to 4.7% in the amplified tests. This is an interesting result showing systematic enhanced coverage. The aggregation of the good results of some languages, such as Python, can be transferred to other less performing languages, such as C.

> **RQ4 insights:** All different setups of *PolyTest* provided gains in the statement coverage and in branch coverage. At best the gains with *PolyTest* in the statement coverage was up to **+7.9%** and in branch coverage was up to **+9.01%**. The minimum gains were less than 0.5% in some cases in *llama3-70b* and *GPT-4o*.

5.5 RQ5

To answer this question, we compute mutation score for the obtained passing tests. Column 8 in Tables 1, 2, 3 gives the average mutation score per language and all *PolyTest* setups for the generated and amplified tests. Once again, for all three LLMs, we observe that the highest coverage metrics are obtained by *PolyTest* (in all its setups) exceeding all other languages in both steps.

For *llama3-70b*, we observe a systematic gain in mutation score up to 8.44% in the generated tests and up to 10.79% in the amplified tests. For *GPT-4o*, we observe a systematic gain in mutation score

up to 10.67% in the generated tests and up to 11.23% in the amplified tests. Finally, for *GPT-3.5*, we observe a systematic gain in mutation score up to 5.14% in the generated tests and up to 10.65% in the amplified tests. This is an important finding that highlights the combined benefits over improved mutation score from unifying tests across different languages and generations. Especially as the mutation score is acknowledge to be a better metric for the tests quality [4, 27, 36].

> *RQ$_5$ insights:* *PolyTest* was able to provide a significant gain in the mutation score up to **+10.79%** with *llama3-70b*, **+11.23%** with *GPT-4o*, **+10.65%** with *GPT-3.5-turbo*.

Table 4: Results for Pynguin generated tests.

Algorithms	n^o of total test	n^o of passing tests	Statement coverage	Branch coverage	Mutation score
DYNAMOSA	441	145	98.68%	98.59%	23.70%
MIO	413	137	98.64%	98.62%	16.82%
Random	12368	5726	98.18%	98.08%	32.54%
whole-suite	961	107	98.76%	98.78%	11.14%

5.6 RQ6

To position *PolyTest* with state of the art test generation, we compare it to a baseline, namely Pynguin tool [30] in Python and its four algorithms DYNAMOSA, MIO, Random, and whole-suite. Overall, Pynguin's results are terrible and do not compete with *PolyTest*.

The number of generated test and passing test is extremely low except for Random algorithm that outperforms *PolyTest*. Only the results of statement and branch coverage are comparable to *PolyTest*, scoring a steady 98% as a minimum. However, on the mutation score, Pynguin shows its weakness. The mutation score varied on average from 11.14% up to 32.54%, which is extremely low compared to *PolyTest*. Finally, we observe that several of Pynguin's generated tests do not contain assertions and are unreadable compared to the *PolyTest* ones. These results demonstrates the benefit of *PolyTest*, in particular, w.r.t. enhancing the tests and their quality.

> *RQ$_6$ insights:* *PolyTest* outperforms Pynguin in generated/passing tests and mutation score, except for coverage.

5.7 Discussion of *PolyTest* impact

Results confirms that *PolyTest*, in all its setups, is effective in enhancing the test suite and its quality, in particular, with the mutation score that is considered a more relevant quality metric than the coverage [4, 27, 36]. It also outperformed the SOTA Pynguin tool in generated Python tests. *PolyTest* ensures to unify the strengths of tests in each single language or each single generation to transfer it to other languages with lower performances. For example, if an LLM is not trained enough on a given language l_1 (e.g., Rust, Swift, Go, etc.) but is trained better on other languages $l_2, l_3, ..., l_n$ (e.g., Python, Java, etc.). One can transfer with *PolyTest* the best results for $l_2, l_3, ..., l_n$ to l_1, either through *PolyTest$_{5\_lang}$*, or through *PolyTest$_{C\times5}$*, *PolyTest$_{CSV\times5}$*, *PolyTest$_{Java\times5}$*, *PolyTest$_{JS\times5}$*,

and *PolyTest$_{Python\times5}$*. It has also the benefit of diversifying the generated an amplified tests. Another advantage of *PolyTest* is its kind of self-consistency in test generation. Indeed, our results demonstrate that the three LLMs generate incorrect contradicting tests, which to the best of our knowledge no prior study has investigated or shown so far. This poses a serious issue if developers are unaware of it. *PolyTest* can detect these contradicting tests to filter and keep the correct ones. In a way, instead of relying on one test output at a time, *PolyTest* has a self-consistency by sampling multiple candidate tests and reconcile them, potentially catching contradictions/errors or omissions/uncovered cases that a single run might miss. Therefore, *PolyTest* benefits developers regardless their chosen programming language and can optimise further the tests quality.

5.8 Threats to validity and limitations

We now discuss internal and external threats to validity [45].

5.8.1 Internal Validity. As we used the EvalPlus dataset that includes canonical solutions, we do not have a threat w.r.t. testing correctness of the tests and we have a full confidence in their results, i.e., passing or failing. We also set temperature at zero for *PolyTest$_{5\_lang}$* so that our experiment and results are deterministic as much as possible, hence, increasing confidence in our results and easing reproducibility. For the other setups of *PolyTest*, we set the temperature to 1 to leverage on its brought diversity and creativity over the 5 generations. We did not further vary the temperature as our goal is to investigate whether unifying tests in multiple languages or generations would improve the quality of the test suite and not the effect of the temperature on it. Thus, studying the effect of varying the temperature.

Moreover, to perform the union, *PolyTest* transforms the tests in different languages to a target language through the LLM itself that. In our experiment the target language was Python and could have been any other language. In fact the target language does not change the results of *PolyTest*, since the unification algorithms will not be impacted. However, the transformation of tests through LLMs raises a risk of mis-translation. To mitigate this, we run random manual checks of translated tests to check their correctness. We found that the translations were correct in all our random verification.

Finally, in our evaluation, we focused solely on LLM-based test generation and only compared against Pynguin [30] since we chose Python as a target language. However, other tools exist, such as EvoSuite [17] for Java, and Nessie [5] for JavaScript. In contrast, *PolyTest* supports multiple languages and can derive tests directly from specifications. Prior evidence suggests that LLM-based test generation can be competitive with, and in some cases even outperform, traditional tools on key metrics [14, 35, 39]. This is confirmed by our comparison with Pynguin in *RQ6*. Our design study demonstrates that leveraging polyglot capabilities and temperature diversity enhances LLM-based test generation by yielding improved coverage and mutation scores. Future research should benchmark against a broader array of test generation tools across various languages to further validate and extend these findings.

5.8.2 External Validity. Our approach is evaluated on three LLMs. Thus, we cannot generalize our results of *PolyTest* on other LLMs,

such as startcoder, etc. However, if another LLM is considered in additional to our three LLMs for the unification over multiple LLMs, at worst *PolyTest* would not decrease the performance observed in our results and at best it would improve them. We further evaluated *PolyTest* over 5 languages only. Other languages, such as Ruby, Rust, etc., could also be considered. Similarly, the results of *PolyTest* at worst would stay the same as in Tables 1, 2, 3 and at best would improve them further. However, we cannot generalize our results for the case of considering other languages and LLMs. This is left for future work. Finally, we evaluated on the dataset of EvalPlus [29] consisting of self-contained functions. Thus, we cannot generalize our results to the more complex programs with dependencies (e.g., program with several interrelated methods) that would require complex objects as input. This is a limitation of our current approach and evaluation that we plan both to adapt and enhance in future work. Nonetheless, by the design of the union of *PolyTest*, the results of a given LLM on a complex program would not worsen and at best would be improved as in our results.

6 RELATED WORK

This section discusses close related work that focuses on empirically evaluating LLMs on test generation. LLMs have been applied in different domains and tasks in Software Engineering [2, 3, 9, 10, 12, 15, 18, 20–23, 28, 32–34, 37, 41, 42, 46–48]. All the above studies focused on either evaluating the ability of LLMs to generate qualitative code, refining it, repairing it if vulnerable, or augmenting it. However, none of them specifically explored the task of test generation. There is only few works assessing the ability of LLMs to generate tests [8, 19, 26, 38–40]. However, to the best of our knowledge, they only evaluate the capability of LLMs to generate tests for a target single language and not their unification.

Indeed, Schafer et al. [39] proposed TESTPILOT, an adaptive LLM-based test generation tool for JavaScript. It relies on GPT3.5-turbo at automatically generates unit tests for the methods in JavaScript projects, evaluated on 25 npm packages. It explores how different prompt components can improve the tests. Siddiq et al. [40] run an empirical study on test generation to compare three LLMs, namely Codex, GPT-3.5-Turbo, and StarCoder, with a focus on test correctness. Baudry et al. [8] focuses on producing fake test data and test data generators with GPT-4 for various application domains in in Chinese, Farsi, Portuguese, Sinhalese, French, Hindi, Spanish, and English. Sapozhnikov et al. [38] introduced TestSpark, a plugin for IntelliJ IDE that enables users to generate unit tests in Java. Li et al. [26] proposed a multi-agent framework called TestChain that decouples the generation of test inputs and test out puts in Python, which gave better results than when generating the tests directly with a 13.84% improvement.

Lemieux et al. [25]proposed to combine Search-based software testing with the Codex LLM to explore whether Codex can be used to help SBST's exploration in Python. This work is interesting as it aims to improve the LLM generated test with SBST. Our work also have similar goal but follows another path by leveraging Multi-langual/Polyglot feature of LLMs. Nashid et al. [31] proposed an approach named CEDAR that create effective prompts to help Codex LLM with different code-related tasks of program repair and test generation by providing example of code and tests. It was evaluated

for two languages. Both Chen et al. [11] and Lahiri et al. [24] used Codex as an LLM to generate code and test cases from problem descriptions in the prompt similarly as in this paper. Bareiss et al. [7] evaluated the performance of Codex on three code-generation tasks, including test generation. They propose embedding contextual information into the prompt to better guide the LLM. El Haji et al. [16] ran an empirical study exploring the effectiveness of GitHub Copilot at generating tests for Python. Gu el al. [19] proposed to improve the LLMs' generated incorrect tests with co-evolution and repair. Pan et al. [35] conducted an empirical study to enhance the LLMs' generated tests with guidance by static analysis demonstrated on Java and Python. Dakhel et al. [14] proposed to enhance the generated tests by including the surviving mutants.

However, all above approaches focus on either investigating the test generation a targeted language or aims to improves its generated tests with repair or static analysis techniques.

Moreover, Wang et al. [44] proposed a strategy of self-Consistency to improve the chain of thought in LLM, which is kind of alternative self-Consistency strategy we use in *PolyTest*. To the best of our knowledge, no study investigated the benefit of leveraging on LLMs diversity induced from their Multi-lingual/Polyglot ability and high temperature over multiple generations to improve the quality of generated and amplified tests. We empirically evaluated how effective this novel strategy implemented in *PolyTest* can improve the quality of the generated and amplified tests.

7 CONCLUSION

In this paper, we presented *PolyTest* a novel approach for the challenge of LLM-based test generation leveraging the the inherent diversity of multilingual/polyglot capabilities and the creative potential of high-temperature sampling. *PolyTest* generates and amplifies a set of tests across multiple languages or through multiple generations, and then unifies these sets, resulting in a significantly consistent and enriched test suite. Our evaluation on three LLMs, namley *LLama3-70b*, *GPT-4o* and *GPT-3.5* and on the EvalPlus dataset showed that *PolyTest* is effective in increasing the size of the test suite and its quality. It improved the obtained tests w.r.t. all metrics we considered, namely number of passing tests, statement and branch coverage, and especially the mutation score that is a better indicator for the tests quality. *PolyTest* also outperformed Pynguin as a baseline comparison. Based on our findings, we recommend the following. First, developers should be aware that generating tests through multiple iterations or across different languages may introduce contradictory test cases. Such inconsistencies can lead to false-positive alerts or create unwarranted confidence in an actually buggy implementation. Second, when working with languages that have weaker support, it is advisable to switch to a more robust language and translate back. This strategy, which is central to *PolyTest*, can mitigate the risk of generating suboptimal test cases. Third, even for well-supported languages, *PolyTest* consistently outperforms single-language or single-repetition solutions without requiring on-the-fly execution for every test case.

For future work, we plan to extend our evaluation with other languages, such as Ruby, Rust, Swift, Go, etc, and to experiment with generating more than five iterations. One limit of our evaluation is the lack of complex programs that we plan to evaluate on in future. However, this will likely require the adaptation of *PolyTest* to take

into account the code implementation of a complex program with its dependencies. This can be trickier to handle than one might think at first glance. Finally, we plan to replicate our study but in other contexts and tasks, such as generation of code solutions, of code patches/repairs, etc. If observed results and gains of *PolyTest* will be observed in other contexts tasks, our methodology could have a deeper impact on the trust or confidence in the LLM results.

REPRODUCTION PACKAGE

Our implementation and evaluation are available in [1].

ACKNOWLEDGMENTS

REFERENCES

[1] [n. d.]. Replication Package. https://anonymous.4open.science/r/Polytest-C96C/. Accessed: 2025-03-14.

[2] Seif Abukhalaf, Mohammad Hamdaqa, and Foutse Khomh. 2023. On Codex Prompt Engineering for OCL Generation: An Empirical Study. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. 148–157. https://doi.org/10.1109/MSR59073.2023.00033

[3] Seif Abukhalaf, Mohammad Hamdaqa, and Foutse Khomh. 2023. On Codex Prompt Engineering for OCL Generation: An Empirical Study. In *20th IEEE/ACM International Conference on Mining Software Repositories, MSR 2023, Melbourne, Australia, May 15-16, 2023*. IEEE, 148–157. https://doi.org/10.1109/MSR59073.2023.00033

[4] James H Andrews, Lionel C Briand, Yvan Labiche, and Akbar Siami Namin. 2006. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering* 32, 8 (2006), 608–624.

[5] Ellen Arteca, Sebastian Harner, Michael Pradel, and Frank Tip. 2022. Nessie: automatically testing JavaScript APIs with asynchronous callbacks. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1494–1505. https://doi.org/10.1145/3510003.3510106

[6] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).

[7] Patrick Bareiß, Beatriz Souza, Marcelo d'Amorim, and Michael Pradel. 2022. Code generation tools (almost) for free? a study of few-shot, pre-trained language models on code. *arXiv preprint arXiv:2206.01335* (2022).

[8] Benoit Baudry, Khashayar Etemadi, Sen Fang, Yogya Gamage, Yi Liu, Yuxin Liu, Martin Monperrus, Javier Ron, André Silva, and Deepika Tiwari. 2024. Generative AI to Generate Test Data Generators. *arXiv preprint arXiv:2401.17626* (2024).

[9] Javier Cámara, Javier Troya, Lola Burgueño, and Antonio Vallecillo. 2023. On the assessment of generative AI in modeling tasks: an experience report with ChatGPT and UML. *Software and Systems Modeling* (2023), 1–13.

[10] Meriem Ben Chaaben, Lola Burgueño, and Houari Sahraoui. 2023. Towards using few-shot prompt learning for automating model completion. In *2023 IEEE/ACM 45th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE, 7–12.

[11] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397* (2022).

[12] K. Chen, Y. Yang, B. Chen, J. Hernandez Lopez, G. Mussbacher, and D. Varro. 2023. Automated Domain Modeling with Large Language Models: A Comparative Study. In *2023 ACM/IEEE 26th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE Computer Society, Los Alamitos, CA, USA, 162–172. https://doi.org/10.1109/MODELS58315.2023.00037

[13] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[14] Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C Desmarais. 2024. Effective test generation using pre-trained large language models and mutation testing. *Information and Software Technology* 171 (2024), 107468.

[15] Jean-Baptiste Döderlein, Mathieu Acher, Djamel Eddine Khelladi, and Benoit Combemale. 2022. Piloting Copilot and Codex: Hot Temperature, Cold Prompts, or Black Magic? *arXiv preprint arXiv:2210.14699* (2022).

[16] Khalid El Haji, Carolin Brandt, and Andy Zaidman. 2024. Using GitHub Copilot for Test Generation in Python: An Empirical Study. (2024).

[17] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 416–419. https://doi.org/10.1145/2025113.2025179

[18] Michael Fu, Chakkrit Tantithamthavorn, Van Nguyen, and Trung Le. 2023. Chat-GPT for Vulnerability Detection, Classification, and Repair: How Far Are We? *arXiv preprint arXiv:2310.09810* (2023).

[19] Siqi Gu, Chunrong Fang, Quanjun Zhang, Fangyuan Tian, and Zhenyu Chen. 2024. Testart: Improving llm-based unit test via co-evolution of automated generation and repair iteration. *arXiv e-prints* (2024), arXiv–2408.

[20] Qi Guo, Junming Cao, Xiaofei Xie, Shangqing Liu, Xiaohong Li, Bihuan Chen, and Xin Peng. 2023. Exploring the Potential of ChatGPT in Automated Code Refinement: An Empirical Study. *arXiv preprint arXiv:2309.08221* (2023).

[21] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2023. Large language models for software engineering: A systematic literature review. *arXiv preprint arXiv:2308.10620* (2023).

[22] Shuyang Jiang, Yuhao Wang, and Yu Wang. 2023. SelfEvolve: A Code Evolution Framework via Large Language Models. *arXiv preprint arXiv:2306.02907* (2023).

[23] Md Mahir Asef Kabir, Sk Adnan Hassan, Xiaoyin Wang, Ying Wang, Hai Yu, and Na Meng. 2023. An empirical study of ChatGPT-3.5 on question answering and code maintenance. *arXiv preprint arXiv:2310.02104* (2023).

[24] Shuvendu K Lahiri, Aaditya Naik, Georgios Sakkas, Piali Choudhury, Curtis von Veh, Madanlal Musuvathi, Jeevana Priya Inala, Chenglong Wang, and Jianfeng Gao. 2022. Interactive code generation via test-driven user-intent formalization. *arXiv preprint arXiv:2208.05950* (2022).

[25] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. 2023. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 919–931.

[26] Kefan Li and Yuan Yuan. 2024. Large Language Models as Test Case Generators: Performance Evaluation and Enhancement. *arXiv preprint arXiv:2404.13340* (2024).

[27] Nan Li, Upsorn Praphamontripong, and Jeff Offutt. 2009. An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage. In *2009 International Conference on Software Testing, Verification, and Validation Workshops*. IEEE, 220–229.

[28] Chao Liu, Xuanlin Bao, Hongyu Zhang, Neng Zhang, Haibo Hu, Xiaohong Zhang, and Meng Yan. 2023. Improving ChatGPT Prompt for Code Generation. *arXiv preprint arXiv:2305.08360* (2023).

[29] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems* 36 (2024).

[30] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2023. An empirical study of automated unit test generation for Python. *Empirical Software Engineering* 28, 2 (2023), 36.

[31] Noor Nashid, Mifta Sintaha, and Ali Mesbah. 2023. Retrieval-based prompt selection for code-related few-shot learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2450–2462.

[32] Nascimento Nathalia, Alencar Paulo, and Cowan Donald. 2023. Artificial Intelligence vs. Software Engineers: An Empirical Study on Performance and Efficiency using ChatGPT. In *Proceedings of the 33rd Annual International Conference on Computer Science and Software Engineering*. 24–33.

[33] Nhan Nguyen and Sarah Nadi. 2022. An empirical evaluation of GitHub copilot's code suggestions. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 1–5.

[34] Ipek Ozkaya. 2023. Application of Large Language Models to Software Engineering Tasks: Opportunities, Risks, and Implications. *IEEE Software* 40, 3 (2023), 4–8. https://doi.org/10.1109/MS.2023.3248401

[35] Rangeet Pan, Myeongsoo Kim, Rahul Krishna, Raju Pavuluri, and Saurabh Sinha. 2024. ASTER: Natural and Multi-language Unit Test Generation with LLMs. *arXiv preprint arXiv:2409.03093* (2024).

[36] Ali Parsai and Serge Demeyer. 2020. Comparing mutation coverage against branch coverage in an industrial setting. *International Journal on Software Tools for Technology Transfer* 22, 4 (2020), 365–388.

[37] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 754–768.

[38] Arkadii Sapozhnikov, Mitchell Olsthoorn, Annibale Panichella, Vladimir Kovalenko, and Pouria Derakhshanfar. 2024. TestSpark: IntelliJ IDEA's Ultimate Test Generation Companion. *arXiv preprint arXiv:2401.06580* (2024).

[39] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering* (2023).

[40] Mohammed Latif Siddiq, Joanna CS Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinícius Carvalho Lopes. 2024. Using Large Language Models to Generate JUnit Tests: An Empirical Study. (2024).

[41] Dominik Sobania, Martin Briesch, and Franz Rothlauf. 2022. Choose your programming copilot: A comparison of the program synthesis performance of github copilot and genetic programming. In *Proceedings of the genetic and evolutionary computation conference*. 1019–1027.

[42] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. 2022. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Chi conference on human factors in computing systems extended abstracts*. 1–7.

[43] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering* (2024).

[44] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-consistency improves chain

[45] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media.

[46] Burak Yetiştiren, Işık Özsoy, Miray Ayerdem, and Eray Tüzün. 2023. Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT. *arXiv preprint arXiv:2304.10778* (2023).

[47] Jiyang Zhang, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. 2023. Multilingual code co-evolution using large language models. *arXiv preprint arXiv:2307.14991* (2023).

[48] Albert Ziegler, Eirini Kalliamvakou, X Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. 2022. Productivity assessment of neural code completion. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. 21–29.

of thought reasoning in language models. *arXiv preprint arXiv:2203.11171* (2022).