Automated Generation of Issue-Reproducing Tests by Combining LLMs and Search-Based Testing

Konstantinos Kitsios
University of Zurich
Zurich, Switzerland
konstantinos.kitsios@uzh.ch

Marco Castelluccio
Mozilla Corporation
London, UK
mcastelluccio@mozilla.com

Alberto Bacchelli
University of Zurich
Zurich, Switzerland
bacchelli@ifi.uzh.ch

***Abstract*—Issue-reproducing tests fail on buggy code and pass once a patch is applied, thus increasing developers' confidence that the issue has been resolved and will not be re-introduced. However, past research has shown that developers often commit patches without such tests, making the automated generation of issue-reproducing tests an area of interest. We propose BLAST, a tool for automatically generating issue-reproducing tests from issue-patch pairs by combining LLMs and search-based software testing (SBST). For the LLM part, we complement the issue description and the patch by extracting relevant context through Git history analysis, static analysis, and SBST-generated tests. For the SBST part, we adapt SBST for generating issue-reproducing tests; the issue description and the patch are fed into the SBST optimization through an intermediate LLM-generated seed, which we deserialize into SBST-compatible form. BLAST successfully generates issue-reproducing tests for 151/426 (35.4%) of the issues from a curated Python benchmark, outperforming the state-of-the-art (23.5%).**

Additionally, to measure the real-world impact of BLAST, we built a GitHub bot that runs BLAST whenever a new pull request (PR) linked to an issue is opened, and if BLAST generates an issue-reproducing test, the bot proposes it as a comment in the PR. We deployed the bot in three open-source repositories for three months, gathering data from 32 PRs-issue pairs. BLAST generated an issue-reproducing test in 11 of these cases, which we proposed to the developers. By analyzing the developers' feedback, we discuss challenges and opportunities for researchers and tool builders.

***Data and material:* https://doi.org/10.5281/zenodo.16949042**

***Index Terms*—test generation, search-based software testing**

I. Introduction

A software issue is typically an online report that describes a bug or a feature request, occasionally including details like stack traces or expected/observed behavior [1]. In most cases, developers address these issues through a set of code changes (also known as a *patch*). In this context, an *issue-reproducing test* is a test accompanying the patch that fails on the unpatched code (validating the presence of the issue) and passes on the patched code (validating its resolution) [2]. An issue-reproducing test increases confidence that the issue (1) can be replicated and (2) will not be reintroduced in the future.

The importance of issue-reproducing tests accompanying a patch is highlighted by previous work [3–6]. For example, Kochhar et al. [3] surveyed 261 developers, asking their view on the statement: "*when a bug is fixed, it is good to add a test that covers it*." Developers agreed with it with a score of 4.4 on a Likert scale [7] of 5, demonstrating the importance of issue-reproducing tests.[1]

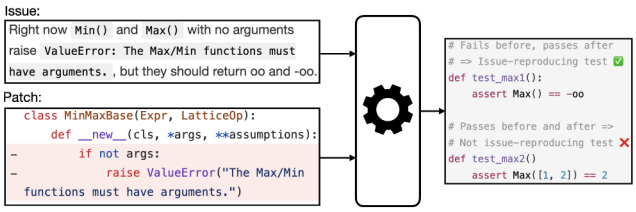

Fig. 1: Overview of the task of automatically generating issue-reproducing tests from issue patches.

Since writing issue-reproducing tests is a demanding and time-consuming task [8, 9] that is often overlooked by developers [10], recent work has investigated ways to automatically generate these tests to support developers. Figure 1 schematically presents this automation task. In particular, recent techniques have proposed the use of Large Language Models (LLMs), either in a zero-shot setting [11] or in a multi-step workflow where the output of the previous LLM call is used as input to the next one [11–15]. For example, AutoTDD [11] feeds the issue description, the patch, and LLM-retrieved code context in an LLM to generate a candidate issue-reproducing test.

However, relying solely on LLMs for the generation of tests is prone to hallucinations [16], leading for example to tests trying to import non-existent modules or use non-existent methods. In addition, recent work does not yet leverage information that could be useful for the generation of the issue-reproducing test such as the focal context or the structure of existing tests for the same class, which could hint the test setup or the available mocks.

To overcome these limitations, we propose BLAST (generating **b**ug-reproducing tests using **L**LMs **a**nd **S**BS**T**) [17], which uses both LLMs and *search-based software testing* (SBST) for the generation of the tests. SBST is an approach to test generation [18] that combines static and dynamic analysis with genetic algorithms to generate tests for a given module that not only are syntactically correct, but also *pass*. BLAST comprises two Components. The SBST Component generates

[1]Reviewers may reject patches until developers add a test: sympy#13647

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00165

candidate issue-reproducing tests by running SBST on the patched module. BLAST encodes information about the issue into SBST by using an LLM to generate a seed test, and then deserializes it into SBST-compatible form. The LLM Component generates a candidate issue-reproducing test by retrieving focal and test context and uses them to build an LLM prompt. BLAST also includes in the prompt the SBST tests generated by its first Component, which are always syntactically correct, passing, and test the changed module.

We evaluate BLAST against a benchmark [11] of 426 issue-patch pairs mined from twelve popular Python repositories and compare it with a zero-shot LLM baseline and the state-of-the-art (SOTA) multi-step LLM workflow [11]. BLAST generates issue-reproducing tests for 35.4% of the issues, outperforming the SOTA (23.5%) using two LLM queries plus one SBST generation[2] instead of three LLM queries.

Evaluating on historical data allows us to run multiple baselines with various settings in a big sample of 426 curated issue-patch pairs, but comes with limitations. In our case, evaluating on historical data (1) is prone to data leakage that could lead to LLM memorization [19, 20] and (2) does not allow us to collect the perception of developers on the generated tests. Other areas evaluating automated methods, such as pull request (PR) reviewer recommendation [21] or code completion [22], employ a broader set of metrics beyond measuring accuracy on historical data to ensure a more comprehensive view of the performance of a system. For example, it was found that models for recommending reviewers can achieve up to 92% accuracy when evaluated against historical data [23], but are not considered useful when deployed with developers [21]. Therefore, we develop and deploy a GitHub bot that runs BLAST whenever a new pull request linked to an issue is opened. When BLAST can generate an issue-reproducing test, the bot proposes it to the developers by leaving a comment in the PR. We deployed the bot to three open-source software repositories for a duration of three months. The bot was triggered in 32 PRs linked to an issue and BLAST generated an issue-reproducing test in 11 of these 32 cases. The developers found the tests valid in 6/11 cases—we discuss their feedback to inform future research and practice.

Our work led to the following main research contributions:

- BLAST, a novel technique combining LLMs with SBST for generating issue-reproducing tests;
- a dataset for evaluating work on SBST for generating issue-reproducing tests, and a semi-manually filtered, higher-quality version of TDD-Bench-Verified;
- evidence of previously unexamined shortcomings of the widely used fail-to-pass metric;
- insights for researchers and practitioners from the first evaluation of an issue-reproducing test generation tool with developers, and the publicly available code of the GitHub bot used in the evaluation.

[2]The SBST generation has lower cost since it only requires a CPU and a user-defined time budget that could be less than a minute.

```
### Module-under-test:
def divide(a: float, b: float) -> float:
    if b == 0:
        raise ValueError("Division by zero")
    return a / b

### Tests generated by Seach-Based Software Testing:
import operations as module_0

def test0():
    float_0 = 10
    float_1 = 2
    assert module_0.divide(float_0, float_1) == 5
```

Fig. 2: Example of a Pynguin-generated test.

II. Background

When generating issue-reproducing tests from issue-patch pairs, the *input* consists of (1) an issue description and (2) a patch that resolves the issue (as shown in Figure 1), and the *output* is an issue-reproducing test. To generate a test, the most widespread and successful approaches in the literature use either SBST or LLMs [13, 24–27]. In the following, we describe these two methods, since BLAST builds on them.

A. Search-Based Software Testing

The goal of SBST is to automatically generate test cases for an input module-under-test (MUT) by covering diverse behaviors [28]. To do so, most SBST tools formulate test generation as an optimization problem: They start with a randomly generated set of test cases, and mutate them to increase coverage. Coverage serves as a fitness function that is maximized using meta-heuristic algorithms such as genetic algorithms [29] and simulated annealing [30]. This optimization continues until the given time budget is exhausted.

Test cases generated by SBST typically consist of sequences of (i) assignments of variables to random values, (ii) method or function (callables) calls, and (iii) assertions that verify the correctness of the call outcomes. The callables are identified by analyzing the MUT source- and byte-code through reflection and static analysis. The expected values of the assertions are generated based on the *observed* behavior during execution, i.e., SBST generates regression assertions [31], guarding against future behavior changes. For this reason, SBST generates *passing* tests for the MUT. Figure 2 shows an example of a test generated by Pynguin [24], the SOTA SBST tool for Python, for a division function.

B. Large Language Models for Test Generation

Pretrained LLMs have shown strong capabilities in generating test code from natural language descriptions [27, 32]. For example, to generate an issue-reproducing test in the zero-shot setting, it is sufficient to feed to an LLM a prompt that contains the issue description and the patch [11]. More recently, workflows consisting of multiple LLM calls have been proposed for code [33, 34] and test generation [11, 13–15]. For example, initial LLM calls are used for action planning [13, 14] or context retrieval [11], and subsequent LLM calls generate and refine the test.

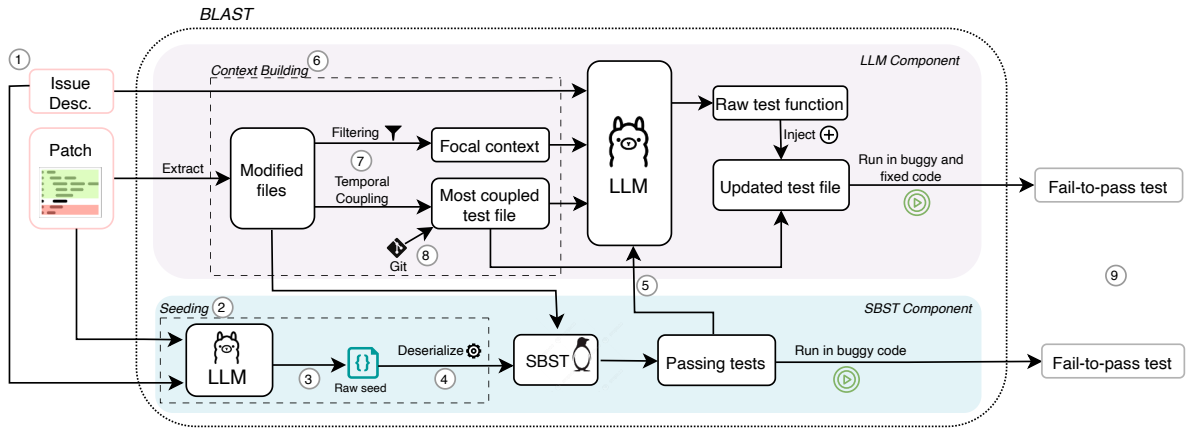

Fig. 3: Overview of BLAST. The SBST Component and the LLM Component coordinate to generate issue-reproducing tests.

III. Related Work

Prior work has proposed methods for generating issue-reproducing tests from issue-patch pairs. SWT-Agent+ [12] is a multi-step LLM workflow where in each step the LLM can read/write/edit files or execute a command, and pass the output to the next step. AutoTDD [11] outperformed SWT-Agent+ using a three-step LLM workflow to select a test file, retrieve relevant context, and generate the test. Also, Ahmed et al. [11] introduce a zero-shot approach with a single LLM query and no context retrieval. These approaches use solely LLMs for the generation of the test and are evaluated only on historical data, which poses memorization concerns and does not take into account the developer's perspective.

Before LLMs, EvoSuiteR [35] adapted EvoSuite, the state-of-the-art SBST tool for Java, for differential testing [36] between two versions of a codebase. However, its lack of scalability [35], the unreadable tests [37], and the lack of encoding of issue information limit its practicality for our task.

Previous work has also explored the adjacent task of generating issue-reproducing tests for test-driven development (TDD) given an issue. LIBRO [2] uses a chain of LLM queries with error feedback to generate issue-reproducing tests for the Defects4J dataset [38]. Otter [14] uses self-reflective action planning with LLMs to plan step-by-step how to generate the test. Issue2Test [13] aims at the generation of tests that fail in the unpatched code through three steps of understanding the issue, generating a candidate test, and refining accordingly. These methods, although effective, focus on the test-driven development scenario of generating an issue-reproducing test before the developer patch is generated, which is not that widely used in practice [39].

Cleverest [15] feeds the commit message and diff to an LLM, prompting it to generate a fail-to-pass input—rather than complete tests—to validate software commits for command-line programs such as XML/PDF parsers. If the generated input is not already a fail-to-pass input, it is used as a seed for the AFL++ fuzzer [40], which mutates it further in an attempt to generate fail-to-pass input.

Finally, Pynguin has been combined with LLMs by CodaMosa [26] for generating passing tests with high coverage. When Pynguin reaches a coverage plateau, CodaMosa uses an LLM-generated seed that helps overcome it. CodaMosa implements a deserialization technique solely based on static rules. We follow a two-step approach, by accompanying the LLM prompt with a manually curated checklist of what a deserialized test looks like, and then applying static rules to filter out the tests where LLM did not follow our checklist.

IV. BLAST: Design and Implementation

This section presents the architecture of BLAST. It takes as input an issue description and the patch resolving that issue (Point 1 in Figure 3) and outputs up to two tests that fail in the unpatched code and pass in the patched one (Point 9). BLAST consists of an SBST Component and an LLM Component, interacting as shown in Figure 3. The SBST Component builds on Pynguin to generate passing tests for the patched module, by starting the mutations from an LLM-generated (Point 3) and deserialized (Point 4) seed to capture the issue semantics. The Pynguin-generated tests are then used as both candidate issue-reproducing tests (Point 9) and additional context for the LLM Component (Point 5). The LLM Component automatically retrieves relevant context (Point 6) from the repository (Point 7), the Git history (Point 8), and the Pynguin-generated tests (Point 5), and feeds it to an LLM to generate a second candidate issue-reproducing test (Point 9).

A. SBST Component

Traditionally, SBST techniques have been devised, evolved, and evaluated with the goal of generating *passing* tests for a given module that maximize coverage [24]. To adapt SBST for generating issue-reproducing tests from issue-patch pairs, BLAST starts the mutations from an LLM-generated and statically deserialized seed that captures the semantics

(e.g., magic values) mentioned in the issue/patch. Out of the resulting SBST-generated tests, BLAST extracts those that reveal differences between the patched and unpatched code. We explain these two mechanisms in more detail below. Since our work focuses on Python, we will use Pynguin [24], the state-of-the-art SBST tool for Python, but BLAST is extensible to work with EvoSuite [28] (Java) as well.

Seeding Pynguin. Pynguin does not accept natural language input, therefore we cannot steer the test generation towards reproducing a given issue. Yet, it is possible to start the mutations from a user-provided seed test.

For this to happen, a technical challenge is that the seed must be a test case in *Pynguin-canonical form*, i.e., the internal representation of tests in Pynguin [24]. In the absence of related documentation, we analyzed the source code of Pynguin and came up with a checklist of properties acceptable in this form. The properties vary from simple ones like "a value should be assigned to a variable before using it as input to a callable", to more complex ones like only importing callables from modules which the MUT imports. The full checklist is available in our replication package [17]. We prompt an LLM to generate a seed given the issue description, the patch, and our curated checklist. The LLM generates a compatible seed in 24% of the cases in our experiments.

Since many seeds are still incompatible, we develop a middleware that statically analyzes the LLM-generated seed to filter out statements that violate the canonical form. Even if a single statement of the seed violates the form, Pynguin will discard the whole seed, so our middleware focuses on detecting statements that violate the form and discarding them so that the rest of the statements can be used. Our middleware increases the seed acceptance rate from 24% to 50%. We note that the other 50% contains cases where a seed cannot be accepted, because to generate a seed related to the issue, the LLM must import code that is not imported in the MUT. In these cases, BLAST runs Pynguin starting from random seeds.

Runtime Filtering of Difference-Revealing Tests. To generate potentially difference-revealing tests, BLAST first extracts the module(s) changed by the patch. Then, it generates tests for *the patched version* of that module(s), which will be, by design, passing. The objective for Pynguin is the whole patched module, as Pynguin does not support finer-grained objectives like the patched function or the patched lines. Finally, BLAST runs the generated tests in the unpatched version, and if a test fails, it is a difference-revealing test, also called *fail-to-pass* (F→P) test. Previous research has shown that SBST may generate flaky tests, i.e., tests that either fail or pass on the same code [41]. To account for flakiness, we run the generated test in the patched code as well, and if it fails (which by design should not), we discard it.

The output of BLAST's SBST Component is (i) a set of tests that pass in the patched code, to be used as prompt context in its LLM Component (Point 5), and (ii) a (potentially empty) set of F→P tests (Point 9).

```
Below is a software issue: <issue>.

A developer resolved the issue with the following patch:
<patch>.

Additional code context is shown below: <focal_code>.

Existing tests for focal code are shown below:
<retrieved_tests>.

Pynguin-generated tests for the focal code are also shown
below: <pynguin_tests>.

Your task is to generate an issue-reproducing test, i.e.,
a test that fails before and passes after the patch.
```

Fig. 4: Outline of the prompt used in the LLM Component of BLAST. The full prompt is available in our replication package.

B. LLM Component

As mentioned in Section II, LLMs can generate tests in a zero-shot setting from a prompt containing the issue description and the patch. However, this context is often not sufficient to capture the complexity of the task, as hinted by the relatively low number (23.5%) of issue-reproducing tests generated by the SOTA [11]. We identify three additional sources of context: existing tests for the MUT, focal context, and SBST-generated tests that pass in the patched code.

Existing Tests Retrieval. Existing tests for the MUT can be useful for two reasons. First, they often contain information on how to set up the test or how to mock objects. Second, the LLM will generate a raw test function, which we must inject into an existing test file. For these reasons, BLAST retrieves the test file most coupled to the changes, and uses it to i) feed its existing tests to the LLM, and ii) inject the generated test.

To infer the most coupled test file, BLAST extracts all the filenames changed in the patch, and applies an initial name-based rule: if `divide.py` is patched, we search for `test_divide.py`. If no such file exists, which could be the case if the tests lived in `test_operators.py`, we propose using *coedit temporal coupling* [42], a metric widely used in the mining software repositories community [43]. Specifically, BLAST iterates all the past commits that changed `divide.py` and searches for the most co-edited file that starts with `test_`. The intuition is that, if whenever `divide.py` changes, `test_operators.py` also changes, then probably the latter contains tests for the former. In some cases, the retrieved file is very large, which caused the LLM performance to drop and made us feed only the first three tests in the prompt.

Dynamic analysis could also be used to infer the most coupled test file, for example by running all the tests and selecting the one that covers more lines of the patch. However, this would require significant compute resources and time, hence we opted for the static heuristic described above that requires far fewer resources.

Focal Context Retrieval. The raw patch contains the lines that changed, which are usually not sufficient for generating an issue-reproducing test. For example, if a method of a class is changed, an issue-reproducing test would generally contain at

least an instantiation of a class object and a call to the changed method. Hence, the LLM should have information about the class signature, the method signatures, the constructor, and information about other properties/methods of the class that could be used in the test setup. This context is also referred to as *focal context* [44]. BLAST retrieves the focal context as follows: if the patch changes a class, it retrieves the class' signature, properties, constructor, and methods. Moreover, BLAST retrieves all the global statements since they often contain useful configurations. If the patch changes a standalone function or expression, BLAST only retrieves the global statements.

Pynguin-Generated Seeds. Finally, BLAST uses the tests generated by its SBST Component as additional context for the LLM. The intuition is that syntactically correct context could reduce the LLM hallucinations, as we qualitatively observe in Section VI. Also, the fact that the tests already pass in the patched code could make the generation of a F→P test easier. The number of tests generated by Pynguin is not constant; a typical range from our experiments is 1 to 50, depending on the size of the MUT. To keep the prompt compact, we feed only the first three tests.

Zero-Shot Prompt. We construct a zero-shot prompt by providing the issue description, the patch, and the retrieved inputs, and asking the LLM to generate a test function that reproduces the issue by failing before and passing after the patch. An outline of the prompt is shown in Figure 4 and is partly inspired by previous work [12]. BLAST feeds the prompt to the LLM, gets back a raw test function, and injects it into the retrieved test as follows: If the test file contains standalone test functions, we inject the generated test function at the end of the file. If, however, the test file contains test classes that in turn contain test methods, we inject the generated test function as a method of the last class of the file, after prepending the argument `self` in the list of arguments. Previous work has experimented with generating a test patch instead of a raw test [12], which would allow editing existing test functions instead of creating new ones and would eliminate the need of injecting the raw test to a file. However, the LLMs hallucinated the line numbers in the patch, leading to many errors [11]. Another alternative is writing each generated test to a new file. This was not preferred because it reduces the practicality, as the developer would need to move the test to the appropriate file. Hence, we opted for asking a raw test function and injecting it to the test file retrieved above.

V. Benchmark-based Evaluation: Methodology

We present here the methodology for the benchmark-based empirical evaluation of BLAST, which we structure around two research questions. Methodological choices specific to each question are presented in Section VI, closer to their results.

A. Research Questions

First, we compare BLAST against two baselines on a recently proposed benchmark of 449 issue-patch pairs across 12 popular Python repositories [11], after applying a semi-manual filtering to discard trivial entries.

RQ1. How does BLAST compare to baselines in generating issue-reproducing tests from issue-patch pairs?

Then, we investigate the contribution of BLAST's LLM and SBST Components. For the former, we perform an ablation study of the input combinations across medium- and large-sized LLMs. For the latter, we study how the LLM seed and the time budget affect the performance and in doing so, we curate and release [17] the first benchmark for SBST on this task.

RQ2. How do the components and hyperparameter choices of BLAST affect its performance?

B. Datasets

As shown in Figure 1, our inputs are an issue description and a patch for this issue. We use the TDD-Bench-Verified (TDD-BV) dataset, which is used in recent literature [11, 14] and consists of 449 issue-patch pairs from 12 open-source Python repositories. Alternative landmark datasets, such as Defects4J [38], have been saturated by recent LLMs [45].

Data Cleaning. TDD-BV is an adaptation of the SWE-Bench dataset [46], which was designed for a different task (i.e., generating a patch given an issue). This difference led to entries like the one shown in Figure 5: Although it is acceptable to

Fig. 5: Example of a trivial instance: Given this issue (#23823 in the `sympy` repository), generating an issue-reproducing test is easy for a modern LLM.

Patch:

```
+ from django.db import DEFAULT_DB_ALIAS
+ ... patch continues to the core logic
```

Pynguin-generated F→P test:

```
def test_case_0():
    // fails in the unpatched code
    assert DEFAULT_DB_ALIAS == 'default'
```

Fig. 6: A F→P test that is not an issue-reproducing test.

use this issue as input to generate the patch, generating an issue-reproducing test given this issue is trivial for modern LLMs, since such a test exists in the input. To refine TDD-BV, we apply a semi-manual filtering process. Specifically, we manually inspect all issues containing the expressions `def test` or `assert`, as these suggest the presence of test code. This inspection yields 37 issues, of which 23 are deemed trivial because they already include a complete issue-reproducing test. In contrast, the remaining issues contain assertions in isolated snippets that merely suggest the test logic, requiring further synthesis by the LLM. We exclude the 23 trivial cases from TDD-BV and use the resulting filtered dataset ($N = 426$) in our experiments.

Pynguin-Compatible Dataset. SBST tools in the literature [18, 26, 28] are evaluated in benchmarks compatible with SBST requirements, since these tools struggle to analyze modules that rely on heavy introspection or dynamic dispatch [24]. Also, only specific versions of Python are supported, namely 3.8–3.10. In the absence of such a benchmark for our task, we set to construct and release PyngBench as follows. We start from the 426 entries of TDD-BV and filter out those using unsupported Python versions, resulting in 254 entries. Then, we set up a Docker container, checkout the patched code, install the appropriate Pynguin version, and run it for one minute to generate tests for the changed modules, following the incompatibility filtering of CodaMosa [26]. The generation is successful in 113 cases, fails due to internal errors in 82 cases (e.g., issue #81), and fails due to incompatible MUT in 172 cases, leaving 113 entries in PyngBench.

We use PyngBench in Section VI-B to study the performance of Pynguin in isolation. In the rest of the paper, BLAST runs in the whole TDD-BV, and when the Pynguin Component fails, BLAST proceeds with the LLM Component.

C. Evaluation Metric: Fail-to-Pass

Our goal is to evaluate to what extent BLAST can generate issue-reproducing tests for a given issue-patch pair. To operationalize the classification of an automatically generated test as an issue-reproducing test, previous work [2, 11–14] has adopted the binary metric *Fail-to-Pass* (F→P). The value of this metric is true *iff* the test fails (F) on the unpatched code and passes (P) on the patched code. The intuition is that an issue-reproducing test should expose the issue and prevent a regression (i.e., fail on the unpatched code) as well as confirm the resolution (i.e., pass on the patched code). In Figure 1, the F→P metric would be evaluated as true[3] for the issue-reproducing test (above), and as false for the other test (below).

Metric Validation. While it is reasonable to think that every issue-reproducing test should be a F→P test, the inverse may not necessarily hold. Consider for example the Pynguin-generated test of Figure 6. The F→P metric is true: the `DEFAULT_DB_ALIAS` is not imported in the unpatched code, so the test fails, but it is imported in the patched code, so the test passes. However, this test does not reproduce the issue, which is about prefetching sliced queries (django #15957).

For these reasons, we empirically validate—for the first time in the literature—how accurately the results of the F→P metric can operationalize the classification of issue-reproducing tests. To this aim, for each F→P test generated by the Pynguin Component of BLAST, we manually examine it to judge whether it is a genuine issue-reproducing test or the metric is evaluated as true for other reasons, such as a side effect in Figure 6. When we refer to the number of F→P tests in the rest of the paper, we mean the manually verified ones for Pynguin, and we use $F \rightarrow P^*$ to indicate all the fail-to-pass tests. The LLM-generated tests are more in number, so we analyze a sample of 50 of them. We find that all of them target the core issue, so for LLM-generated tests we assume $F \rightarrow P^* = F \rightarrow P$. This is because, in contrast to Pynguin, the LLM has explicit access to the issue, which, under our current prompt, makes it generate true issue-reproducing tests. Given our limited expertise with the 12 repos, we have high confidence for the absence of obvious cases like in Figure 6 but we may have missed borderline cases, which we tackle by evaluating F→P tests with developers in Section VII-B.

D. Experimental Setup

We ran Pynguin on an Ubuntu 22.04 machine with 16 Intel Xeon CPUs and 128GB RAM, using the random seed 42 to increase reproducibility. We use three LLMs, namely gpt-4o (gpt-4o-2024-08-06), llama3.3 (llama-3.3-70b-versatile), and DeepSeek (deepseek-r1-distill-llama-70b), covering a range of medium- to large-sized models, with and without reasoning. For gpt-4o and llama3.3 we use a temperature of 0 to increase reproducibility and also follow related work [11, 14]. We use the latest version of Pynguin, which is v0.41 at the time of writing. For instances of TDD-BV that run on Python < 3.10 (and are therefore not supported in Pynguin v0.41), we use the most recent version of Pynguin that still supports Python < 3.10 (i.e., Pynguin v0.17). We keep the default Pynguin parameters (e.g., `DYNAMOSA` algorithm, $crossover\,rate = 0.75$, $population = 50$), unless explicitly specified otherwise in our ablation study (Section VI-B).

VI. Benchmark-based Evaluation: Results

We present further methodological details for each research question and report the corresponding results.

[3]Henceforth, we refer to a test for which the *Fail-to-Pass* (F→P) metric is evaluated to be true as a *Fail-to-Pass test* or *F→P test*.

TABLE I: Performance of BLAST against two baselines.

LLM	Method	# F→P	% F→P
gpt-4o	ZeroShot	86	20.2
	AutoTDD	100	23.5
	BLAST (Ours)	**151**	**35.4**
llama3.3	ZeroShot	45	10.6
	AutoTDD	51	12.0
	BLAST (Ours)	**131**	**30.8**
DeepSeek	ZeroShot	43	10.1
	AutoTDD	48	11.3
	BLAST (Ours)	**114**	**26.8**

TABLE II: Input combinations (C_1–C_7) and their effect on BLAST's LLM Component (underlying model in gpt-4o).

	Input Combinations					Results (F → P)		
ID	Issue	Patch	Focal	Ex. Tests	P. Tests	#	%	Cum. %
C_1	✓			✓		84	19.7	19.7
C_2		✓	✓			86	20.2	32.6
C_3	✓	✓				124	29.1	42.3
C_4	✓	✓	✓			137	32.2	45.1
C_5	✓	✓	✓	✓		140	32.9	48.8
C_6	✓	✓	✓		✓	**145**	**34.1**	49.3
C_7	✓	✓	✓	✓	✓	143	33.6	49.5

A. RQ1 - Performance

To demonstrate the effectiveness of BLAST, we compare it to two baselines from the literature. AutoTDD [11] achieves SOTA results in generating issue-reproducing tests from issue-patch pairs using the three-step LLM workflow described in Section III. ZeroShot is a single-step baseline from Ahmed et al. [11] that feeds the issue description and the patch to the LLM, without any additional retrieved context.

Table I presents the results: BLAST outperforms previous methods across all underlying LLMs evaluated. The best performance is with gpt-4o, where AutoTDD generates issue-reproducing tests in 100/426 (23.5%) of the issues, while BLAST does sp for 151/426 (35.4%). Regarding overlap, BLAST generates issue-reproducing tests for 67 issues that AutoTDD misses, while AutoTDD does so for 16 issues that BLAST misses.

Finding 1. BLAST could generate issue-reproducing tests for 151/426 of the cases—an absolute increase of 51 cases over the state-of-the-art (100/426).

BLAST differs from AutoTDD in two key aspects: (1) the augmented context provided in the LLM prompt, and (2) the SBST Component. In RQ2, we analyze the contribution of each of these aspects to the performance improvement.

We also report the patch coverage [47], i.e., the patch lines that are covered by the F→P tests. The coverage ranges from 91% for ZeroShot to 94% for BLAST but the difference is not statistically significant. Previous work has also observed that for F→P tests the coverage is high regardless of the generation method [11, 12], so the comparison between two methods is reduced to the comparison of their F→P rate.

Regarding the cost of each approach, Pynguin achieves the best result with a time budget of 60 seconds. We assume that an LLM query and the context extraction in both BLAST and AutoTDD have zero time overhead; In reality, these processes take up to a few seconds, which is negligible compared to the 60 seconds. Under these assumptions, BLAST requires two LLM queries and 60 seconds time overhead, while AutoTDD requires three LLM queries with no time overhead.

Finally, to evaluate how well BLAST retrieves the most related test file, we compare the retrieved file with the file where the developer placed their test in the actual PR. BLAST's name- and Git-based heuristic matches the developer file in 86% of the cases, compared to AutoTDD's LLM-based retrieval which only matches the developer's file in 61% of the cases.

B. RQ2 - Ablation

As shown in Figure 3, BLAST consists of an LLM Component and an SBST Component, each generating a candidate test. We investigate how the two Components and their hyper-parameters contribute to the result of Table I.

LLM Component. To assess the impact of additional prompt information on the performance of the LLM Component, we begin with the baseline prompt described in Section IV and construct seven variants, each including only a specific subset of the prompt inputs. Table II details the configurations of these seven input combinations. The first combination (C_1) corresponds to the test-driven development case (as investigated in previous work [2, 11–14]) where the test must be written before the patch and only the issue description is available with potential example tests. In C_2, we only provide the issue title without description to simulate scenarios where the issue description is absent or very poor, as may happen in practice [1]. C_3 uses only the issue description and the patch which are readily available, and we use it to measure the performance increase with our focal code retrieval (C_4), test retrieval (C_5), and SBST-generated tests (C_6, C_7). We run BLAST with each input combination C_i and report the results for our best model gpt-4o in Table II, while the results for the other models exist in our replication package [17].

For the TDD scenario (C_1), BLAST generates a F→P test in 19.7% of the cases even though it was not designed for TDD. When only the issue title is given instead of the description (C_2), BLAST performs significantly worse (8.9%) than when the description is added (C_3). This demonstrates the relevance of a good issue description for test generation tools. By adding focal context (C_4), the performance increases by another 3.1%. The retrieved test case (C_5) yields small increase (0.7%) in gpt-4o, but yields the largest increase for llama3.3 (6.3%). Incorporating Pynguin-generated tests into the prompt (C_6, C_7) yields the best performance (+1.2), even though Pynguin-generated tests were only available for 113 out of the 426 cases. DeepSeek and llama3.3 follow a similar trend with small differentiations that we analyze in our replication package [17], with gpt-4o outperforming them across all C_i.

Unexpectedly, the cumulative number of F→P tests generated by the union of all combinations (last column of Table II) is 15.4% higher than the best-performing individual run. We

TABLE III: Effect of Pynguin variations (N=113).

Variation	#F→P*	#F→P	% F→P	#F→P Unq.	% F→P Un
$t = 6$	10	7	6.2	4	3.5
t = 60	**13**	**10**	**8.8**	**6**	**5.3**
$t = 600$	12	9	7.9	5	4.4
No seed	8	4	3.5	3	2.7

attribute this to the fact that LLMs have been shown prioritize content towards the end of the prompt, known recency bias [48]. So, when we add for example the existi tests at the end of the prompt (C_5), the focal code that is earlier in the prompt now, receives less attention.

SBST Component. We run a set of experiments disabling BLAST's LLM Component to measure the performance of the SBST Component and explore the impact of various parameters. For the methodological reasons explained in Section V-B, we run this set of experiments against the PyngBench dataset.

To analyze the impact of time budget on performance, we run Pynguin with three configurations: the default $t = 600$ seconds commonly used in prior studies [25, 26], a practical $t = 60$ seconds suitable for real-time GitHub bot usage, and a minimal $t = 6$ seconds to approximate the latency of a typical LLM query. To evaluate the contribution of our seeding mechanism (Point 2 in Figure 3), we also run Pynguin without seeding while fixing $t = 60$. All other parameters are set to their default values, following past studies [26]. Table III presents the results; the last two columns report the number and percentage of issue-reproducing tests uniquely generated by the SBST Component, i.e., cases in which the LLM Component failed to produce an issue-reproducing test.

With a 60-second time budget, Pynguin generates an F→P test in 13/113 cases; however, our manual inspection revealed that only 10 are true issue-reproducing tests, while the others are similar to the case shown in Figure 6. Surprisingly, extending the time budget to 600 seconds results in one less F→P test. Through manual analysis, we found that Pynguin initially generated tests for the focal method `nthroot_mod()`, but after additional mutations, it discovered `sqrt_mod()` and `primitive_root()`, which achieved higher overall coverage, thus it discarded tests involving `nthroot_mod()`. This illustrates a key limitation of Pynguin: it optimizes for module-level coverage and does not focus on a specific focal method—a limitation we further discuss in Section VI-C.

The last row of Table III reports the results of running Pynguin for 60 seconds without our seeding mechanism (Point 2 in Figure 3), thus initializing mutations from purely random tests. In this setting, only four F→P tests are generated, highlighting the critical role of our seeding approach in aligning Pynguin's search with the issue description and the patch.

Overall Contribution of SBST. One of our contributions is the introduction of SBST for generating issue-reproducing tests through a two-way orchestration between SBST and LLMs: SBST generates tests that are used as (i) candidate issue-reproducing tests (Point 9 in Figure 3), and (ii) prompt context for an LLM that will generate candidate issue-reproducing tests (Point 5). From Table III we see that Pynguin generates issue-reproducing tests for 6 entries where the LLM failed, and from Table II we see that including the Pynguin-generated tests in the LLM prompt leads to 5 additional issue-reproducing tests, for a total of 11. Hence, Pynguin contributes in 7.3% (11/151) of the generated issue-reproducing tests of BLAST. In the following, we present two exemplary cases out of these eleven.

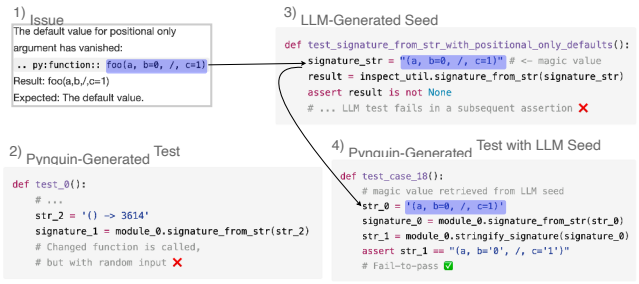

Fig. 7: Issue-reproducing test generated by Pynguin with gpt-4o seeds.

Finding 2. BLAST's LLM Component generates an issue-reproducing test in 151/426 cases, with the focal context and the SBST tests being the most important prompt context. BLAST's SBST Component generates an issue-reproducing test in 10 cases, 5 of which missed by the LLM Component. Overall, SBST contributes to 11/151 (7.3%) of the successful cases.

C. Analysis Of Success And Failure Cases

We showcase the effective coordination between the two Components of BLAST through two representative examples. Additionally, to examine BLAST's limitations, we manually analyzed 50 cases in which it failed to generate an issue-reproducing test.

Representative Success Cases. Consider the issue of Figure 7. For this issue, Pynguin without a seed generated the test shown in the bottom-left of the figure, where the patched function (`signature_from_str()`) is correctly called, but with a random input. By starting the mutations from the LLM seed (top-right), the value required to reproduce the issue (highlighted in blue) is used instead of a random string, leading to the issue-reproducing test shown in the bottom-right.

Pynguin-generated tests can also serve as prompt context for the LLM, as we demonstrate with the issue of Figure 1, for which gpt-4o generated the test in the top-left of Figure 8. This test throws an error instead of passing in the patched code, because the LLM considers `-oo` to be a singleton that, similar to `oo`, can be imported. In reality, `-oo` is the result of the `__neg__` method of the `oo` singleton and as such cannot be imported. For the same issue, Pynguin generated the test shown in the bottom-left of Figure 8, which is not a F→P test, but it uses the negative infinity properly, by importing

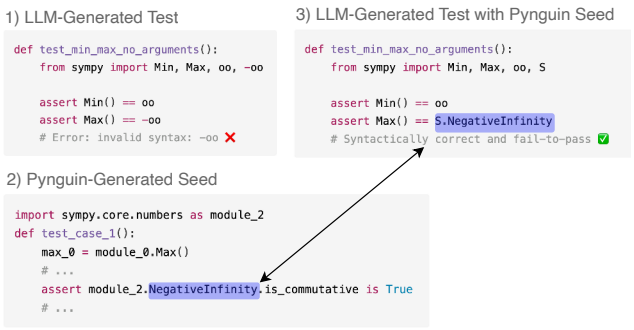

Fig. 8: Issue-reproducing test generated by gpt-4o with Pynguin tests as prompt context.

`sympy.core.numbers.NegativeInfinity`. When this syntactically correct test was included in the prompt, the LLM generated the issue-reproducing test shown in the top-right of Figure 8, which is almost the same as the top-left one but with `NegativeInfinity` instead of `-oo`.

Analysis of Failure Cases. We manually analyze 30 cases where BLAST's LLM Component failed to generate an issue-reproducing test and 20 where the SBST Component failed and categorize the failure reasons using open card sorting [49].

Regarding the LLM Component, the failure is due to mistakes in the LLM code in 22/30 cases, which we categorize in *logical* (12), caused by an assertion error (or lack thereof), and *syntactic* (10), where the LLM could not set up the test. In only two of the syntactic failures further context could help, while for the other eight cases no context is missing: in two such examples, the LLM defines a helper function with two arguments and calls it later without arguments.

In five of the other eight cases, the LLM does *not comply with requested output format*. For example, we ask for a raw test function and it returns a `unittest.TestCase` class, which our injection is not designed to handle. This happens in Django issues because Django uses the `unittest` framework, so gpt-4o defaults to generating a unittest.TestCase despite instructed otherwise. Finally, in three cases the issue is *missing metadata*: the python version (the LLM code calls a deprecated method), installed pip packages (the LLM imports a non-installed package), and the exact date when the test is run (the test must use a future date, and the LLM uses 2023, which is not the future anymore).

Regarding the SBST Component, the most common issue (10/20) is that the tests do not cover the changes, because Pynguin generates tests for whole modules and not classes or methods. In 5/20 cases, the test requires setup currently unsupported by Pynguin, e.g., the patch only adds decorators to methods, and Pynguin does not support decorators. In 3/20 cases, the changes are covered but not triggered, while in 2/20 cases Pynguin generates tests that require additional setup to run, leading to errors. We release the analyzed examples along with our observations in our replication package [17].

TABLE IV: Mozilla repositories used in our study.

Repository	Description	# stars	# forks	# PRs
bugbug	Platform for ML on SE	537	311	26
bugbot	Mozilla release management tool	48	71	5
libmozdata	Library to access Mozilla data	12	13	1

VII. In Vivo Evaluation

Despite increasing research on the automated generation of issue-reproducing tests, the evaluation of these techniques in real-world development environments is lacking. Such evaluations are crucial for at least two reasons: First, existing benchmarks—including those based on SWE-Bench [11, 12]—rely on historical data, in some cases dating back to 2013, thus raising risks of LLM memorization [19]. Second, the ultimate goal of automation is to assist developers; therefore, developer perception and output usability are to be taken into account. Ignoring this dimension risks painting a partial picture, as illustrated in the domain of reviewer recommendation: although some models achieved up to 92% accuracy on historical benchmarks [23], these models proved not useful when deployed in real development settings [21].

We conduct a real-world evaluation of BLAST by deploying it in three open source repositories for three months, allowing us to both evaluate BLAST in a memorization-free setting and collect developers' perceptions on the generated tests.

A. Evaluation Methodology

We develop and open-source a GitHub bot that, once deployed in a repository, is triggered when a PR is opened. Then, if the PR (i) resolves an issue, (ii) edits at least one `.py` file, and (iii) does not accompany the patch with a test, the bot retrieves the patch and the issue and feeds them into BLAST. If BLAST generates a F→P test, our bot proposes it to the developer as a PR comment. This allows us to evaluate BLAST's performance in a memorization-free setting, collect developers' perceptions of the tests, and surface challenges for researchers and practitioners based on developers' feedback. We focus on the following research questions:

RQ3.1. How does BLAST perform in a real-world setting?
RQ3.2. How do developers perceive the issue-reproducing tests it generates?

We deployed the bot for three months across three open-source repositories within the Mozilla corporation, as detailed in Table IV, collecting data from 32 pull requests (PRs) in total. For each PR, we executed the LLM Component of BLAST once for each of the three LLMs of Table I, to compare their performance and maximize the number of tests eligible for developer feedback. To avoid overwhelming contributors—which prior work suggests may reduce the quality of feedback [50]—we proposed only a single test per PR. When both Components of BLAST produced an issue-reproducing test for the same PR, we prioritized the LLM-generated test, due to previous work suggesting that SBST tests are often

TABLE V: PRs for which at least a F→P test was generated. ✓ denotes an issue-reproducing test; ❍ denotes a F→P that is not an issue-reproducing test; ✓ denotes a developer-accepted test; ✗ denotes that Pynguin could not run in this PR.

Comp.↓ / PR →		1	2	3	4	5	6	7	8	9	10	11
LLM	gpt-4o	✓	✓	✓			✓		✓	✓		
	llama			✓	❍		✓	❍	✓	✓	❍	❍
	DeepSeek			✓	❍	❍			✓			❍
Pynguin		✗	✗	✗	✗				❍	❍	✗	

rejected by developers due to reduced readability [37]. In cases where multiple LLMs generated an issue-reproducing test, we selected the test from the higher-performing LLM in Table I. All the F→P tests for a given PR that were not proposed to the developer, were manually assessed by the first author to determine whether they are issue-reproducing tests.

For each proposed test, we ask the maintainer(s) of the repos to provide feedback on the test in two steps. First, we ask if they consider the proposed F→P test to be an issue-reproducing test, followed by an open answer to the following question: "*Reason for not accepting the test, or minor changes you would require to accept it, or general comment if you will accept it as is.*" We give the option to request minor changes, because previous work has shown that when proposing automatically generated tests to open-source maintainers, they often request such changes [51].

In total, we propose tests in 11 out of 32 PRs, each represented as a column in Table V, where the rows represent the two Components of BLAST. The F→P tests that the developer judged to be issue-reproducing are denoted with ✓, while the rest are marked with ❍. The issue-reproducing tests that were also merged in the project are shown with ✓. Finally, PRs for which Pynguin could not run successfully are denoted with ✗. We discuss the developer feedback in the next section.

B. Results

In contrast to our benchmark-based evaluation, llama3.3 achieves the highest number of F→P tests (8/32, 25.0%), followed by gpt-4o (7/32, 21.9%) and DeepSeek (4/32, 12.5%). The smaller number of parameters of llama3.3 could lead to less memorization than larger models, and hence, lower performance on historical data. Gpt-4o has been shown to memorize more than llama3.3 in non-SE tasks [20], however, a larger sample is needed to study this difference in more detail.

Pynguin successfully ran in 11/32 PRs and generated two F→P tests, but our bot did not propose them to the developer because gpt-4o-generated tests had precedence. After inspecting the tests ourselves, we found that they were not issue-reproducing tests. The 11 PRs where Pynguin ran successfully is a small sample, and since SBST tools do not suffer from memorization, Pynguin's performance on the historical data is expected to be the same as in new data.

The above suggest that BLAST can generate issue-reproducing tests in a real-world setting, beyond benchmarks of historical, potentially memorized data. The accuracy of BLAST in the real-world setting is lower than in TDD-Bench-Verified, but we cannot attribute the drop to memorization only, since the repositories are also different.

Finding 3.1. BLAST generated fail-to-pass tests in 8/32 (25%) PRs when deployed in three open-source repos, with llama3.3 outperforming larger models.

We analyze here the developer feedback for the 11 proposed tests. The developer considered 6/11 (55%) tests as valid issue-reproducing tests, two of which were successfully included in the projects' test suite, one without any changes and the other after inserting it in a different test file. In one case the test was not added because the PR was closed without merging, while in another case, the developer said that the test reproduces the issue, but "it does not test a critical part," i.e., the issue was not critical enough to require a test. Finally, in two cases the developer deemed the test to be reproducing the issue, but with excessive mocking that missed important aspects of the implementation. Specifically, the PR patched a crash caused by the field `comments` missing from a data structure. To patch this bug, the developer created the function `handle_missing_comments()` to add the field if missing. The proposed test set up the data with the missing field, but simply asserted that `handle_missing_comments()` was called, while it should also check that the field exists.

We analyze below the 5/11 PRs in which the developer judged the F→P test as *not* issue-reproducing. In two PRs, the issue was a renaming of a component from `Fenix` to `Firefox for Android`; BLAST generated a test that asserts the new name, but for such patches, an accompanying test was not necessary. In two other cases, the issue was requesting a new feature and the maintainer explained: "The test would be perfect if this were a regression, but it is a feature." Upon a follow-up question, the maintainer clarified that tests for new features are generally useful, but for these specific two they were not that useful. Finally, the last PR resolves a bug where a query incorrectly ignored some entries, by changing the query parameters. In the lack of context for a proper issue-reproducing test, BLAST asserts that the query parameters have changed, which is not reproducing the issue.

The above feedback highlights key distinctions between SWE-Bench-based evaluations and real-world deployments. In SWE-Bench, a relevant test is guaranteed to exist for each patch because SWE-Bench consists of patches that i) resolve an issue, *and* ii) include a developer-written test. In contrast, our evaluation focuses on the utility of automatically generated tests *in the absence of developer-written ones*, which also resulted in the bot proposing tests even when no tests were relevant for the patch. To mitigate these false positives, one developer suggested limiting bot activation to issues labeled as "bug" or allowing developers to invoke the bot on demand.

Another developer explained that the proposed tests become obsolete when new commits are pushed to the PR, and suggested triggering the bot on each new commit, which we implemented. We note, however, that for multi-revision PRs,

this could swamp the comment section. An on-demand bot could also help in this case.

Finding 3.2. Of the 11 proposed tests, 6 were considered valid issue-reproducing tests and 2 were integrated in the test suite. Reasons for not considering the tests are that, in contrast to benchmarks, some issues do not require an accompanying test and that the test uses excessive mocking, missing core functionality.

VIII. Discussion

In this section, we discuss the implications of our findings for future research and practice, along with the limitations of BLAST and the threats to the validity of our study.

On Benchmarks and Metrics. We underline the importance of manually inspecting benchmarks, as our analysis of TDD-BV revealed 23 trivial issues that could inflate performance. Furthermore, as a community we should reflect on evaluation metrics like F→P, which, while widely used, do not always correspond to issue-reproducing tests. Manual validation, developer validation, and matching the failure trace to the issue, as done by Issue2Test [13], are possible ways forward.

Recommendations for Practitioners. For developers deploying test generation tools, we recommend being aware of cases where a patch does not require a test and syncing with developers to agree upon the tool *triggering criteria*. PRs often undergo many revisions, thus generating one test per revision may overwhelm the developers. Other options like generating whenever a PR is marked as "ready" could be beneficial. On-demand tools may address both challenges, though they shift some responsibility to developers, potentially reducing automation.

Recommendations for Researchers. While improving upon the SOTA, BLAST manages to generate an issue-reproducing test for 35.4% of the issue-patch pairs in the benchmark and for 25.0% in the real-world evaluation, which leaves ample room for future improvement. We hope that our manual analysis of 50 issues where BLAST failed can provide directions for future research.

BLAST provides evidence that going beyond only LLMs for test generation can be beneficial, as indicated by the contribution of SBST in 7.3% of the issue-reproducing tests. However, the contribution of SBST is relatively low, which we trace back to two main *limitations*. The first limitation is the inability of SBST tools to run in arbitrary code, e.g., Pynguin runs successfully in 113/426 (26%) dataset instances. If future Pynguin versions or new SBST tools add support for more complex code, the contribution of Pynguin-generated tests would subsequently increase. The second limitation is the inability of SBST tools to generate tests for specific methods or lines instead of whole modules, which leads to SBST-generated tests not covering the patch in 10/20 failure cases manually analyzed in Section VII-B. More advanced SBST tools that support test generation for fine-grained targets like functions or lines would mitigate this limitation and further improve the contribution of SBST.

Threats to Validity. Our study includes two manual inspection steps: the filtering of TDD-BV for trivial entries, and the judgment of whether a F→P test is an issue-reproducing test, and as such these inspections could be prone to human error. We release the inspected entries to be independently validated.

Our experiments focus on Python and use SWE-Bench-derived benchmarks, hence our results may not generalize to other languages or codebases. The real-world study includes 32 PRs, which limits generalization, but provides a solid first step towards evaluating similar tools with developers.

Both LLMs and Pynguin are non-deterministic systems. To boost reproducibility, we used a fixed seed for Pynguin, but running on different hardware could result in slightly different results. To mitigate the non-determinism of LLMs we used a temperature of zero and pinned the versions of each model. However, repeating each experiment multiple times, with different seeds for Pynguin, would have further strengthened the statistical power of our experiments, and the absence of such repetitions is a potential threat to validity.

The authors of the two baselines could not provide the replication package at the moment, so we implemented the approaches based on their paper. Our implementation closely matches the reported performance (23.5% vs 24.3%) using the same pinned version of gpt-4o.

IX. Conclusion

In this paper, we presented BLAST, a hybrid approach that combines LLMs and SBST to generate issue-reproducing tests. By statically retrieving and generating context for the LLM and adapting SBST for generating issue-reproducing tests, BLAST outperforms the state-of-the-art, achieving 35.4% success rate. We evaluated BLAST in a widely used benchmark of historical data to understand how different design decisions affect its performance. By further evaluating BLAST in vivo, we increased our confidence about its ability to perform in memorization-free environments, and also gathered developer feedback that could be helpful for future research and practice.

Acknowledgments

K. Kitsios and A. Bacchelli gratefully acknowledge the support of the Swiss National Science Foundation through the SNSF Project 200021_197227.

References

[1] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?" in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2008, pp. 308–318.

[2] S. Kang, J. Yoon, and S. Yoo, "Large language models are few-shot testers: Exploring llm-based general bug reproduction," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2312–2323.

[3] P. S. Kochhar, X. Xia, and D. Lo, "Practitioners' views on good software testing practices," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019, pp. 61–70.

[4] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal, "A study of effective regression testing in practice," in *PROCEEDINGS The Eighth International Symposium On Software Reliability Engineering*. IEEE, 1997, pp. 264–274.

[5] A. K. Onoma, W.-T. Tsai, M. Poonawala, and H. Suganuma, "Regression testing in an industrial environment," *Communications of the ACM*, vol. 41, no. 5, pp. 81–86, 1998.

[6] S. Wang, X. Lian, D. Marinov, and T. Xu, "Test selection for unified regression testing," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1687–1699.

[7] R. Likert, "A technique for the measurement of attitudes." *Archives of psychology*, 1932.

[8] A. Labuschagne, L. Inozemtseva, and R. Holmes, "Measuring the cost of regression testing in practice: A study of java projects using continuous integration," in *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, 2017, pp. 821–830.

[9] P. Straubinger and G. Fraser, "A survey on what developers think about testing," in *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2023, pp. 80–90.

[10] S. Levin and A. Yehudai, "The co-evolution of test maintenance and code maintenance through the lens of fine-grained semantic changes," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 35–46.

[11] T. Ahmed, M. Hirzel, R. Pan, A. Shinnar, and S. Sinha, "Tdd-bench verified: Can llms generate tests for issues before they get resolved?" *arXiv preprint arXiv:2412.02883*, 2024.

[12] N. Mündler, M. Müller, J. He, and M. Vechev, "Swt-bench: Testing and validating real-world bug-fixes with code agents," *Advances in Neural Information Processing Systems*, vol. 37, pp. 81 857–81 887, 2024.

[13] N. Nashid, I. Bouzenia, M. Pradel, and A. Mesbah, "Issue2test: Generating reproducing test cases from issue reports," *arXiv preprint arXiv:2503.16320*, 2025.

[14] T. Ahmed, J. Ganhotra, R. Pan, A. Shinnar, S. Sinha, and M. Hirzel, "Otter: Generating tests from issues to validate swe patches," *arXiv*, 2025. [Online]. Available: https://arxiv.org/abs/2502.05368

[15] J. Liu, S. Lee, E. Losiouk, and M. Böhme, "Can llm generate regression tests for software commits?" *arXiv preprint arXiv:2501.11086*, 2025.

[16] A. Eghbali and M. Pradel, "De-hallucinator: Mitigating llm hallucinations in code generation tasks via iterative grounding," *arXiv preprint arXiv:2401.01701*, 2024.

[17] K. Kitsios, M. Castelluccio, and A. Bacchelli. (2025, Aug.) Replication package for "automated generation of issue-reproducing tests by combining llms and search-based testing". [Online]. Available: https://doi.org/10.5281/zenodo.16949043

[18] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Computing Surveys (CSUR)*, vol. 45, no. 1, pp. 1–61, 2012.

[19] A. Hooda, M. Christodorescu, M. Allamanis, A. Wilson, K. Fawaz, and S. Jha, "Do large code models understand programming concepts? counterfactual analysis for code predicates," in *Forty-first International Conference on Machine Learning*, 2024.

[20] N. Cohen-Inger, Y. Elisha, B. Shapira, L. Rokach, and S. Cohen, "Forget what you know about llms evaluations – llms are like a chameleon," 2025. [Online]. Available: https://arxiv.org/abs/2502.07445

[21] V. Kovalenko, N. Tintarev, E. Pasynkov, C. Bird, and A. Bacchelli, "Does reviewer recommendation help developers?" *IEEE Transactions on Software Engineering*, vol. 46, no. 7, pp. 710–731, 2018.

[22] P. Vaithilingam, T. Zhang, and E. L. Glassman, "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*, 2022, pp. 1–7.

[23] V. Balachandran, "Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 931–940.

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

[25] S. Lukasczyk, F. Kroiß, and G. Fraser, "An empirical study of automated unit test generation for python," *Empirical Software Engineering*, vol. 28, no. 2, p. 36, 2023.

[26] 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 *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 919–931.

[27] G. Ryan, S. Jain, M. Shang, S. Wang, X. Ma, M. K. Ramanathan, and B. Ray, "Code-aware prompting: A study of coverage-guided test generation in regression setting using llm," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 951–971, 2024.

[28] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 416–419.

[29] J. H. Holland, *Adaptation in natural and artificial sys-*

tems: an introductory analysis with applications to biology, control, and artificial intelligence. MIT press, 1992.

[30] S. Kirkpatrick, C. D. Gelatt Jr, and M. P. Vecchi, "Optimization by simulated annealing," *science*, vol. 220, no. 4598, pp. 671–680, 1983.

[31] T. Xie, "Augmenting automatically generated unit-test suites with regression oracle checking," in *European Conference on Object-Oriented Programming*. Springer, 2006, pp. 380–403.

[32] V. Guilherme and A. Vincenzi, "An initial investigation of chatgpt unit test generation capability," in *Proceedings of the 8th Brazilian Symposium on Systematic and Automated Software Testing*, 2023, pp. 15–24.

[33] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "Autocoderover: Autonomous program improvement," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1592–1604.

[34] I. Bouzenia, P. Devanbu, and M. Pradel, "Repairagent: an autonomous, llm-based agent for program repair.(2024)," *arXiv preprint arXiv:2403.17134*, 2024.

[35] S. Shamshiri, "Automated unit test generation for evolving software," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 1038–1041.

[36] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[37] G. Grano, S. Scalabrino, H. C. Gall, and R. Oliveto, "An empirical investigation on the readability of manual and generated test cases," in *Proceedings of the 26th Conference on Program Comprehension*, 2018, pp. 348–351.

[38] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 international symposium on software testing and analysis*, 2014, pp. 437–440.

[39] A. Causevic, D. Sundmark, and S. Punnekkat, "Factors limiting industrial adoption of test driven development: A systematic review," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 2011, pp. 337–346.

[40] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "{AFL++}: Combining incremental steps of fuzzing research," in *14th USENIX workshop on offensive technologies (WOOT 20)*, 2020.

[41] M. Gruber, M. F. Roslan, O. Parry, F. Scharnböck, P. McMinn, and G. Fraser, "Do automatic test generation tools generate flaky tests?" in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–12.

[42] C. Gote, I. Scholtes, and F. Schweitzer, "Analysing time-stamped co-editing networks in software development teams using git2net," *Empirical software engineering*, vol. 26, no. 4, p. 75, 2021.

[43] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," in *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. IEEE, 1998, pp. 190–198.

[44] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, and N. Sundaresan, "Unit test case generation with transformers and focal context," *arXiv preprint arXiv:2009.05617*, 2020.

[45] D. Ramos, C. Mamede, K. Jain, P. Canelas, C. Gamboa, and C. L. Goues, "Are large language models memorizing bug benchmarks?" *arXiv preprint arXiv:2411.13323*, 2024.

[46] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. R. Narasimhan, "SWE-bench: Can language models resolve real-world github issues?" in *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: https://openreview.net/forum?id=VTF8yNQM66

[47] M. Hilton, J. Bell, and D. Marinov, "A large-scale study of test coverage evolution," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 53–63.

[48] A. Peysakhovich and A. Lerer, "Attention sorting combats recency bias in long context language models," *arXiv preprint arXiv:2310.01427*, 2023.

[49] D. Spencer, *Card sorting: Designing usable categories*. Rosenfeld Media, 2009.

[50] B. Fass-Holmes, "Survey fatigue–what is its role in undergraduates' survey participation and response rates?." *Journal of interdisciplinary Studies in Education*, vol. 11, no. 1, pp. 56–73, 2022.

[51] C. Brandt, A. Khatami, M. Wessel, and A. Zaidman, "Shaken, not stirred. how developers like their amplified tests," *IEEE Transactions on Software Engineering*, 2024.