Evaluating LLM-Based Regression Test Generation

JING LIU*, MPI-SP, Germany
SEONGMIN LEE*, University of California at Los Angeles, USA
ELEONORA LOSIOUK, University of Padua, Italy
MARCEL BÖHME, MPI-SP, Germany

Large Language Models (LLMs) have shown tremendous promise in automated software engineering. In this paper, we investigate the opportunities of LLMs for just-in-time regression test generation for programs, like parsers, interpreters, or compilers, that take highly structured, human-readable inputs. When a new bug fix or code change is committed, the repository (as part of the CI/CD workflow) runs an LLM for a few minutes to generate regression test cases for that commit that exercise the changed code and potentially trigger any bugs.

Specifically, we investigate LLM-based regression test generation as a *machine translation task* that takes the developer-provided commit message, the code change, and the name of the input format (e.g., XML) and produces regression test cases for the described change in the given input format. In our experiments testing 72 commits to Mujs, Libxml2, Poppler, JerryScript, Z3, PHP, JQ, and MicroPython, our feedback-directed, zero-shot LLM-based prototype Cleverest performed well, even if we did *not* provide the code change. In under 2 minutes, on average, Cleverest found as many bugs as the state-of-the-art directed greybox fuzzer WAFLGo in 24 hours, even though WAFLGo started with a commit-reaching seed corpus in the majority of cases. If we amplify the Cleverest-generated test cases using those as a seed corpus in coverage-guided greybox fuzzing, the number of bugs found doubles. We call the integration with fuzzing as ClevFuzz.

In addition, we found that some commit messages are more expressive than others, thus we wonder how this impacts the effectiveness of Cleverest. Our results above demonstrate that Cleverest picks up on the change intention. For instance, if the commit message describes that this patch changes how floating point variables are treated in the Mujs javascript interpreter, then Cleverest generates Javascript programs that contain floating point variables. To study the impact of expressiveness, we change the commit messages minimally to reduce and increase the information in the commit message, respectively, and find a substantial impact on effectiveness. For instance, adding 17 words on average (max. 43) to make ineffective commit messages more expressive significantly increased the number of bugs found.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; *Software evolution.*

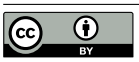
ACM Reference Format:
Jing Liu, Seongmin Lee, Eleonora Losiouk, and Marcel Böhme. 2026. Evaluating LLM-Based Regression Test Generation. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE122 (July 2026), 23 pages. https://doi.org/10.1145/3808129

1 Introduction

Recently, Large Language Models (LLMs) have shown tremendous promise. Liu et al. [18] published a comprehensive survey of research on the automation of software engineering processes, including requirements engineering, code generation, program analysis, testing, debugging, and end-to-end development and maintenance.

*Both authors contributed equally to this research.

Authors' Contact Information: Jing Liu, MPI-SP, Germany, jing6@acm.org; Seongmin Lee, University of California at Los Angeles, USA, seongminlee@sigsoft.org; Eleonora Losiouk, University of Padua, Italy, eleonora.losiouk@unipd.it; Marcel Böhme, MPI-SP, Germany, marcel.boehme@acm.org.



ACM 2994-970X/2026/7-ARTFSE122
https://doi.org/10.1145/3808129

In this paper, we explore the utility of LLMs as test generators for code commits or pull requests in the continuous integration/continuous development (CI/CD) pipeline. When a new bug fix or a code change is committed and the project is built and analyzed as part of CI/CD pipeline, running an LLM for a few minutes to generate regression test cases for that commit could help exercise the changed code and potentially trigger any bugs. Specifically, discovering bugs i) introduced or ii) insufficiently fixed by a commit are the two primary objectives of our paper. We investigate LLM-based regression test generation as a *machine translation task* that takes the developer-provided commit message, the code change, and the name of a well-known input format (e.g., XML or JS) and produces regression test cases for the commit in the given input format. In systems without precise test oracles (e.g., interpreters, parsers, compilers), such regressions are typically identified via crashes, sanitizer violations, or observable output differences, like in (differential) fuzzing.

In our experiments, we tested 72 commits to eight programs (Mujs, Libxml2, Poppler, JerryScript, Z3, PHP, JQ, and MicroPython), which take highly structured, human-readable input formats. Our feedback-directed, zero-shot LLM-based prototype Cleverest performed well. In under two minutes, on average, Cleverest found as many bugs as the state-of-the-art directed greybox fuzzer WAFLGo found in 24 hours, even though WAFLGo [38] already starts with a commit-reaching seed corpus for the majority of commits. We find that our tool's effectiveness varies across input formats and LLMs. For instance, Cleverest performs much better for human-readable formats like XML and JavaScript than for binary formats like PDF. Effectiveness also scales with LLM size: a weaker `GPT-4o mini` reduces effectiveness, while a stronger `DeepSeek-R1` found a bug missed by `GPT-4o`.

Most interestingly, we identify a critical dependence of Cleverest's effectiveness on the commit message. In fact, Cleverest continues to generate effective test cases even when only the commit message is provided, as long as the intention of the change is captured. For instance, if the commit message describes the handling of floating point variables, then Cleverest generates Javascript programs that contain floating point variables. In contrast, the non-descriptive commit messages in Z3 (e.g., fix #[Issue ID]) are unlikely to yield effective regression tests while the expressive commit messages are more likely to reveal bugs. To study the impact of expressiveness, we change the commit messages minimally to reduce and increase the information, respectively, and indeed confirm a substantial impact on effectiveness. For instance, adding 17 words on average (max. 43) to make ineffective messages more expressive significantly increased the number of bugs found.

We also explore the marriage of greybox fuzzing and Cleverest, called ClevFuzz, where the regression tests generated by Cleverest are provided as initial seeds for subsequent greybox fuzzing. Indeed, if we amplify the Cleverest-generated test cases in this way, we can find over twice the bugs as the directed fuzzer WAFLGo [38] with its default seed and Cleverest alone.

In summary, this paper makes the following contributions:

- We introduce Cleverest, a feedback-directed, zero-shot LLM-based regression test generation technique for programs that take highly structured, human-readable inputs to evaluate the utility of LLMs for regression test generation.
- We evaluate Cleverest on 72 commits to eight popular programs and find that Cleverest performs very well for programs using human-interpretable file formats, even when only the commit message is given. This might mean, the LLM can translate the change *intention* into a system input. We confirm this hypothesis by varying the expressiveness of the commit message.
- Cleverest performs comparably to WAFLGo, a SOTA few-shot regression test generator, while being substantially faster—even though WAFLGo's initial seeds are already close to bug-revealing. Using Cleverest-generated tests as seeds for fuzzing finds more than twice as many bugs.
- All implementation code, data, and scripts are available in our replication package.

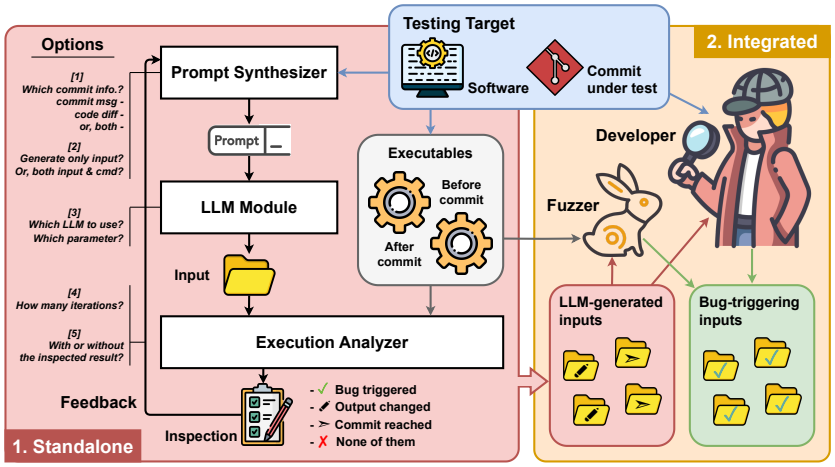


Fig. 1. CLEVEREST: Our zero-shot feedback-guided LLM-based regression test generation methodology.

2 Experimental Design

In this paper, we seek to evaluate the capabilities of a large language model (LLM) as a regression test generation tool in the CI/CD pipeline for programs that take highly structured, human-readable inputs, like JavaScript programs or XML files. While classical regression testing focuses on passing test cases across program versions, our setting lacks a specification or pre-existing tests. Thus, our primary objective is to determine how well an LLM performs in generating test cases that: reach the changed code, observe behavioral differences, and trigger bugs detectable via sanitizers. These metrics reflect standard practice in regression fuzzing, where a regression manifests as a newly introduced crash. We implement a feedback-directed LLM-based regression test generator involving a prompt synthesizer and execution analyzer. We call our tool CLEVEREST.

2.1 Research Questions

RQ1. *How well does CLEVEREST perform as a regression test generator?* We evaluate the two most important properties of CLEVEREST if it was fully embedded in a CI/CD pipeline: effectiveness and execution time. Specifically, we evaluate how well CLEVEREST can find bugs that were introduced in a commit (*bug-finding*) and how well CLEVEREST can reproduce bugs that were patched in a commit (*bug-reproduction*). To understand how CLEVEREST performs in the hands of a developer, we evaluate how "close" CLEVEREST-generated test cases are to revealing a bug—both quantitatively, using execution feedback distance, and qualitatively, through manual inspection.

RQ2. *How well does CLEVEREST perform under various hyperparameter values? (Ablation Study)* Specifically, we analyze how well it performs compared to the default configuration if we (a) only used the commit message but not the diff, (b) only used the commit diff but not the message, (c) let it generate the command in addition to the input, (d) used maximum LLM temperature, (e) used a less or a more powerful LLM (`GPT-4o mini` vs `DeepSeek-R1`), (f) used 10 feedback loop iterations instead of 5, and (g) dropped the execution analysis in the feedback loop.

RQ3. *How informative and self-contained do commit messages have to be to enable regression test generation without access to code?* Specifically, we investigate the impact of varying the descriptiveness of commit messages on CLEVEREST's performance; our preliminary results from **RQ2** suggest that even with commit messages alone, CLEVEREST can generate effective test cases.

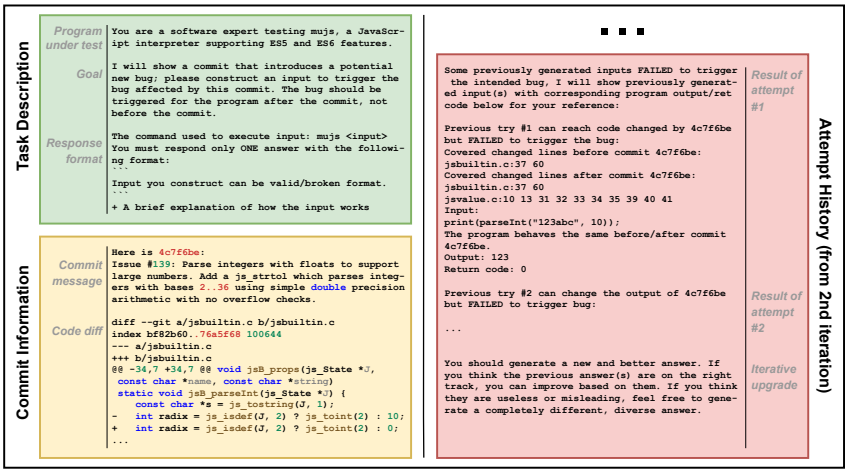


Fig. 2. The prompt generated by the prompt synthesizer for the LLM to generate the test input for Mujs.

RQ4. *How does* Cleverest *compare to WAFLGo, the state-of-the-art in regression test generation?* WAFLGo [38] is the state-of-the-art directed greybox fuzzer for regression test generation that was shown to outperform previous directed and regression greybox fuzzers [4, 8, 19, 40, 41]. Specifically, we evaluate (1) the performance of WAFLGo as a few-shot regression test generator and (2) the dependence of WAFLGo on the initial seeds, and (3) the performance of Cleverest against WAFLGo both directly and as a seed generator for zero-shot greybox fuzzing.

More results. Separately, we compare our LLM-based regression test generation approach with an *undirected* LLM-based test generation approach, where no change information is provided—similar to existing methods explored in previous work [9, 22, 37]. Moreover, we evaluate the performance of Cleverest on the 50 most recent commits in Libxml2. For both, we postpone the results to the **Supplementary Material** [17] for space reasons.

2.2 Cleverest: LLM-based Regression Testing Technique and Implementation

Figure 1 gives a procedural overview of Cleverest, our zero-shot feedback-guided LLM-based regression test generation tool. It consists of three key components: *Prompt Synthesizer*, *LLM Module*, and *Execution Analyzer* that work in a *feedback loop* to generate test cases for a commit.

2.2.1 Prompt Synthesizer. The *Prompt Synthesizer* generates a domain-specific prompt for the LLM to create the test input. Given the testing target (i.e., the software and the commit under test) and optionally some feedback from a previous iteration, the prompt synthesizer constructs the prompt for the LLM, which includes the task description, commit information, and the attempt history.

Figure 2 shows an example prompt to generate a regression test case for a code commit to Mujs, a lightweight JavaScript interpreter. The synthesized prompt has the following structure:

(1) **Task Description**: The prompt starts with a general description of the task. The main parts consist of (a) a simple sentence to describe the program under test, e.g., "JavaScript interpreter," (b) the goal of the LLM-generated input, e.g., "trigger a bug introduced by the commit," and (c) the format specification of the LLM's response.

(2) **Commit Information**: The second part contains the commit under test. We consider both the commit message and the code diff as part of the commit information, unless otherwise stated.

The *code diff* contains the changed lines plus a few lines of context. The *commit message* offers a high-level description of the developer's intention and the purpose of the commit.

(3) **Attempt History**: When available, the prompt contains the execution result of the previously generated test case. Each item shows (a) the general execution result from the execution analyzer (cf. Sec. 2.2.3), (b) the program output (incl. `stdout` & `stderr`), and (c) the return code.

Hyperparameters. The prompt synthesizer can be configured differently affecting the task difficulty for the LLM. For instance, we may consider providing only the code diff, only the commit message, or both. We may provide the command-line utility and parameters (`cmd`) expected to be used to test the commit or let it generate itself: Notice that fuzzing can only modify the input and requires the user to specify the `cmd`. Yet, an LLM can generate the full `cmd` for the program execution. Allowing the generation of the `cmd` can make the task more challenging for the LLM, as it needs to understand the program's behavior to generate the correct `cmd`.

2.2.2 LLM Module. The generated prompt is fed into the LLM module, which generates the test input for the commit under test. The output of the LLM contains the content of the test input along with the explanation of the expected behavior and the `cmd` if asked. The output is then parsed to produce the actual test case, which is executed to verify the commit under test. Depending on the task description, the LLM tries to generate the input that triggers the bug introduced by the commit (without knowing a priori if the bug is introduced) or reproduces the bug fixed by the commit.

The LLM module has several hyperparameters that affect the quality of the generated output. One is the *size* (or capacity) of the LLM model: A larger model size generally results in better output quality but requires more computational resources. Another is the *temperature*, which controls the randomness of the output: A higher temperature results in more randomness in the output.

2.2.3 Execution Analyzer and Feedback Loop. The *Execution Analyzer* receives the LLM-generated test input and executes it on the program before and after the commit under test. The execution analyzer compares the two executions to measure if the generated input contributes to the commit testing. We measure the effectiveness using an ordinal scale:

✓ **Bug Triggering**: To discover a bug if one exists is the ultimate objective of testing. Depending on the scenario, the execution analyzer checks if the input triggers the intended bug: it should trigger the bug after or before the commit in the bug-finding or bug-reproduction scenario, respectively, and should not trigger the same bug in the other program version. For instance, even if the input triggers a bug, for our intents and purposes it is not considered as bug triggering if the same bug is triggered in both versions of the program.

✎ **Output Changing**: Even if the input does not trigger a bug (e.g., if the sanitizer does not detect the anomaly), the input may witness a difference in behavior. We detect the behavior difference by comparing the output and return code of the program before and after the commit. Any difference will imply that the generated test successfully exposes the difference in the program behavior introduced by the commit.

➢ **Commit Reaching**: The first necessary condition for an input to reveal a bug introduced by the commit is to reach the code changed by the commit. Yet, even generating an input that reaches the commit is not trivial. We thus measure the coverage of the program execution with the generated input to see if there is any overlap between the code changed by the commit and the code covered by the execution.

The output of the execution analyzer is the assessment result of the generated input categorized into four types: bug triggered, output changed, commit reached, and none of the above. Notice that the prior categories are strict inclusions of the latter ones.

If the execution results do not trigger a bug, the inspected execution result (i.e., which lines of the commit the input covers if at all, and whether the output is different) is provided to the prompt synthesizer for the next input generation. This incremental prompting strategy guides the LLM in exploring the input space more effectively and allows it to generate high-quality test input for the commit under test. This is also confirmed by recent works [7, 10, 33] that showed LLM-based test generation benefits substantially from feedback loops that augment prompts with analysis results.

Two hyperparameters in the feedback loop affect the performance of CLEVEREST. One hyperparameter allows to enable or disable the feedback loop. If disabled, only the previously generated input is used as the prompt for the next iteration to avoid generating the same input. The other hyperparameter is the number of iterations, which determines the number of times the LLM generates the input, and the execution analyzer verifies the input. A higher number of iterations can lead to more effective input generation but requires more computational resources.

2.2.4 Implementation. For *commit information*, we use Git to extract the commit message and code diff. The command '`git show --format=%B`' is used to print them together in a concise format. We remove all changes that are not related to the source code.

To *detect whether a bug is triggered*, our approach relies on an automated failure oracle that flags failures when executing generated test cases. Any mechanism that can automatically identify failures—such as sanitizers, assertions, or runtime checks—can be integrated without changes to the test generation pipeline. In this study, we use AddressSanitizer (ASan) [29], which inserts instrumentation during compilation to detect memory-corruption at runtime, including buffer overflows and use-after-free errors, without requiring manually written oracles. To *detect behavior differences*, we compare the `stdout`, `stderr`, and return code of the program built before and after the commit under test. To *detect whether the changed code is reached*, we use `GCOV` to get code coverage information. We consider the input reaches the commit if there is any overlap between the code changed by the commit and the code covered by the execution.

For our *LLM interactions*, we use `openai-cli`, a command-line client. We use `GPT-4o` as the default model and set the max token to 4096 to achieve a balance between performance and cost. Notably, we do not keep conversation history when querying LLM, as it consumes more tokens. The default temperature is set to 0.5, and the number of iterations is set to 5. We additionally consider up to ten iterations and the temperature to be 1.0 to examine parameter effects. We also consider two other models, `GPT-4o mini` and `DeepSeek-R1`, to evaluate the impact of model size on performance. `GPT-4o mini`, a smaller and more efficient version of `GPT-4o`, which is designed especially for applications where affordability and lower latency are critical. `DeepSeek-R1`, recently released (Jan. 2025), is the cutting-edge LLM model especially designed for reasoning tasks. It has 671B parameters and scores 90.8% on the MMLU benchmark. As a reasoning model, `DeepSeek-R1` returns both the final model output and intermediate reasoning traces, but, since they are returned separately, we exclusively retrieved the final generated test input and ignored the intermediate reasoning output. This strategy allowed us to prevent excessively long reasoning chains from affecting the correctness or structure of the generated regression test cases.

2.3 Benchmark Selection and Infrastructure

To answer our research questions, we choose a commit benchmark dataset for our experiment based on the following *selection criteria*:

(1) We want to focus on regression test generation for programs that take highly structured, human-readable input formats.[1]

[1]Without seeds or grammars this type of programs pose a challenge for existing tools while LLMs are known to handle highly structured text well. We are sure to make claims only with respect to this type of programs.

Table 1. Detail of Bug Dataset. The three last columns show the ID in the issue tracker and the commit IDs of the bug-introducing-commit (BIC) and the bug-fixing-commit (BFC).

Software	Description	Command-Line Utility and Parameters	Issue	BIC	BFC
Mujs	JavaScript interpreter	`mujs` ⟨input file⟩	#65	8c27b12	833f82c
		`mujs` ⟨input file⟩	#141	832e069	6871e5b
		`mujs` ⟨input file⟩	#145	4c7f6be	f93d245
		`mujs` ⟨input file⟩	#166	3f71a1c	8b5ba20
Libxml2	XML parsing library	`xmllint --recover --sax1 --sax` ⟨input file⟩	#535	9a82b94	d0c3f01
		`xmllint --recover --dropdtd --sax1 --nofixup-base-uris` ⟨input file⟩	#550	74c84a8	6273df6
		`xmllint --encode UTF-16 --maxmem 325 --encode UTF-16` ⟨input file⟩	#553	1406b20	db21cd5
		`xmllint --htmlout` ⟨input file⟩	#720	14604a4	3ad7f81
		`xmllint --repeat --push` ⟨input file⟩	#841	9cfc723	dc6270d
Poppler	PDF rendering library	`pdfunite` ⟨input file 1⟩ ⟨input file 2⟩ `output.pdf`	#1282	3d35d20	4564a00
		`pdfunite` ⟨input file 1⟩ ⟨input file 2⟩ `output.pdf`	#1289	3cae777	efb6868
		`pdftops` ⟨input file⟩ `/dev/null`	#1303	e674ca6	a4ca3a9
		`pdftoppm -mono -cropbox` ⟨input file⟩	#1305	aaf2e80	907d05a
		`pdftoppm -mono -cropbox` ⟨input file⟩	#1381	245abad	1be35ee
JerryScript	JavaScript interpreter	`jerry` ⟨input file⟩	#5013	b7e3bae	3686410
		`jerry` ⟨input file⟩	#5138	70e275e	95ec5e9
		`jerry` ⟨input file⟩	#5117	53b61c1	d7e2125
		`jerry` ⟨input file⟩	#5153	70e275e	de51531
Z3	Theorem prover	`z3` ⟨input file⟩	#6659	0f3c562	a849a29
		`z3` ⟨input file⟩	#6914	a62e4b2	eea9c0b
		`z3` ⟨input file⟩	#7246	dec87fe	49610f5
		`z3` ⟨input file⟩	#7252	a7b1dbc	a6b5027
PHP	PHP interpreter	`php` ⟨input file⟩	#16777	5cb38e9	18b18f0
		`php` ⟨input file⟩	#16978	8c704ab	d17ed34
		`php` ⟨input file⟩	#17442	998bce1	b068c2f
		`php` ⟨input file⟩	#17463	7904a08	e4473ab
JQ	Command-line JSON processor	`jq -nf` ⟨input file⟩	#2825	680baef	3fa10e8
		`jq .` ⟨input file⟩	#2976	680baef	71c2ab5
		`jq .[54E100]=7` ⟨input file⟩	#3262	0e067ef	de21386
		`jq -nf` ⟨input file⟩	#4901[2]	4003202	499c91b
MicroPython	Python for microcontrollers	`micropython` ⟨input file⟩	#13007	4c93955	f397a3e
		`micropython` ⟨input file⟩	#13041	1fef566	908ab1c
		`micropython` ⟨input file⟩	#17733	5e9189d	3185bb5
		`micropython` ⟨input file⟩	#17815	69ead7d	c0252d7
		`micropython` ⟨input file⟩	#17841	e702046	4b013ec
		`micropython` ⟨input file⟩	#17847	7e14680	0615d13

(2) We want to maximize the fairness of our comparison to the state-of-the-art regression test generation tool WAFLGo, which is used for comparison.
(3) We consider real-world regression bugs of various types that have been introduced and patched in identified commits to open-source software.

Table 1 details the dataset that meets our selection criteria. Specifically, we found that the benchmark dataset used in WAFLGo's evaluation [38] satisfies criteria (2) and (3), while three programs (the JavaScript interpreter Mujs, the XML parsing library Libxml2, and the PDF rendering library Poppler) satisfy criterion (1). Additionally, we added five more subject programs—the JavaScript interpreter JerryScript, the SMT solver Z3, the PHP interpreter PHP, the JSON processor JQ, and the Python interpreter MicroPython—which are frequently used in software testing papers [30, 37]. These programs cover various input formats. Each software has one or more command-line utilities that retrieve textual input files as command-line parameters with relevant options.

The WAFLGo dataset lists 11 memory-related bugs for these three benchmark programs. For the five newly added programs, we searched GitHub issues and selected the four to five newest reproducible bugs for each. We also added three extra bugs for Libxml2 to make it also have five bugs. The bug types span a range of memory-related issues, including buffer overflows and use-after-frees. For each bug, we intentionally include both bug-introducing commits (BICs) and bug-fixing commits (BFCs). BICs are used to evaluate the *bug-finding* scenario, modeling realistic feature or refactoring changes that may introduce regressions. BFCs are used to evaluate the *bug-reproduction* scenario, representing cases where no regression test existed prior to the fix; the objective is to automatically synthesize a regression test that would prevent re-introduction of the fixed bug. In total, we have 72 commits in the experiment.

Experiment Configuration. The default configuration of our CLEVEREST is as follows: We provide the LLM with both the commit message and code diff as commit information, along with the command-line utility and parameters used to test the commit. Execution analysis results are included in feedback. We use the `GPT-4o` model with 0.5 temperature to balance creativity and precision, following ChatAFL [21]. We limit each run to a safe upper bound of five feedback iterations based on our preliminary study, where we observed that effective test cases were generated within 1.67 iterations on average. Each LLM-based input generation has a 30-second timeout.[3] To account for randomness in the LLM and fuzzer, we repeat each experiment ten times. All experiments run in a Docker container with 64 AMD EPYC 7713P cores @ 2.0GHz and 251GB memory.

[2] JQ #49014 refers to CVE-2025-49014, which is publicly available on GitHub security advisories but not issue tracker.
[3] Due to free-tier API limitations, DeepSeek-R1 experiments had a median execution time of approximately 15 minutes; commercial deployments would substantially reduce this latency.

Table 2. Results for bug finding and bug reproduction effectiveness of CLEVEREST across ten repetitions. For each bug and scenario, we report the average effectiveness score on a slider (✘: Not reached; ➢: Reached; ✎: Output-changing; ✓: Bug-revealing), the number of trials where the commit was reached (Reach), output was changed (Change), bug was found (Bug), and the average time in seconds (T (s)).

		Bug-Introducing-Commit					Bug-Fixing Commit				
		✘ ➢ ✎ ✓	Reach	Change	Bug	T (s)	✘ ➢ ✎ ✓	Reach	Change	Bug	T (s)
Mujs	#65		10/10	10/10	10/10	7.5		10/10	6/10	6/10	19.1
	#141		10/10	10/10	0/10	45.4		10/10	10/10	1/10	31.7
	#145		10/10	8/10	8/10	37.1		10/10	9/10	9/10	28.8
	#166		10/10	1/10	0/10	57.6		10/10	10/10	10/10	7.4
Libxml2	#535		10/10	10/10	10/10	5.4		10/10	10/10	10/10	5.5
	#550		10/10	2/10	0/10	27.6		0/10	0/10	0/10	29.9
	#553		10/10	10/10	10/10	6.7		10/10	10/10	10/10	13.7
	#720		10/10	0/10	0/10	53.6		7/10	0/10	0/10	104.1
	#841		10/10	10/10	10/10	5.9		10/10	10/10	10/10	5.5
Poppler	#1282		8/10	3/10	0/10	196.6		9/10	0/10	0/10	123.5
	#1289		4/10	0/10	0/10	165.0		8/10	6/10	6/10	90.8
	#1303		0/10	0/10	0/10	218.7		1/10	1/10	0/10	132.6
	#1305		0/10	0/10	0/10	267.2		0/10	0/10	0/10	147.7
	#1381		0/10	0/10	0/10	202.8		0/10	0/10	0/10	147.1
JerryScript	#5013		10/10	0/10	0/10	106.8		10/10	0/10	0/10	72.0
	#5117		10/10	10/10	0/10	71.4		10/10	5/10	5/10	50.8
	#5138		10/10	10/10	2/10	78.7		10/10	0/10	0/10	74.2
	#5153		10/10	10/10	0/10	95.2		0/10	0/10	0/10	39.4
Z3	#6659		0/10	0/10	0/10	136.2		0/10	0/10	0/10	74.7
	#6914		2/10	0/10	0/10	188.9		0/10	0/10	0/10	76.3
	#7246		10/10	0/10	0/10	75.6		0/10	0/10	0/10	68.6
	#7252		8/10	0/10	0/10	100.6		5/10	0/10	0/10	76.1
PHP	#16777		10/10	0/10	0/10	109.9		10/10	0/10	0/10	87.7
	#16978		10/10	1/10	0/10	106.0		0/10	0/10	0/10	70.7
	#17442		10/10	10/10	0/10	103.3		0/10	0/10	0/10	78.4
	#17463		6/10	0/10	0/10	122.7		10/10	10/10	10/10	9.1
JQ	#2825		10/10	0/10	0/10	23.6		10/10	1/10	0/10	28.4
	#2976		10/10	0/10	0/10	22.3		10/10	10/10	10/10	3.8
	#3262		10/10	10/10	10/10	9.7		10/10	10/10	10/10	8.4
	#49014		2/10	2/10	0/10	32.2		0/10	0/10	0/10	32.7
MicroPython	#13007		9/10	7/10	0/10	27.0		1/10	1/10	0/10	26.6
	#13041		0/10	0/10	0/10	31.0		10/10	10/10	0/10	21.8
	#17733		10/10	0/10	0/10	26.3		10/10	0/10	0/10	31.4
	#17815		10/10	9/10	0/10	26.3		10/10	10/10	10/10	5.5
	#17841		10/10	10/10	0/10	21.4		10/10	0/10	0/10	39.2
	#17847		10/10	10/10	0/10	40.9		0/10	0/10	0/10	29.2
Average			**31/36**	**20/36**	**7/36**	**79.3**		**25/36**	**17/36**	**13/36**	**52.6**

3 Experimental Results

RQ1. Evaluation of Capabilities

Effectiveness. We measure the effectiveness of CLEVEREST by computing the *average effectiveness score* across all ten repetitions. If the generated regression test case finds the bug in the bug-introducing commit (BIC) or reproduces the bug in the bug-fixing commit (BFC), the score is 3 (✓). Similarly, 2 indicates an output-changing test case (✎), 1 indicates a commit-reaching test case (➢), and 0 indicates a test case that fails to reach the changed code (✘). Visually, we represent the average effectiveness score on a slider. The slider is colored in red, orange, olive, and teal if the score is within the range of [0-0], (0-1], (1-2], and (2-3], respectively.

Table 2 shows how well CLEVEREST performs as a regression test generator. We find that CLEVEREST performs well in bug finding and bug reproduction for four programs—Mujs, Libxml2, JerryScript, and JQ —that use more human-interpretable formats. Despite lacking access to the full program code and without explicit information about the input features needed to exercise the changed code, CLEVEREST identified bugs in 7 out of 17 BICs and reproduced the bugs patched in 10 out of 17 BFCs. Furthermore, in 14 out of the remaining 17 commits of these four programs, it at least reached the changed code. It also performs meaningfully well for PHP and MicroPython, reaching the changed code in 16 out of 20 commits and reproducing bugs in 2 BFCs. CLEVEREST shows consistent high reachability for these two interpreter programs, yet bug-triggering can be harder due to their complex language features. For example, in PHP #16777, CLEVEREST successfully reached the vulnerable logic by re-invoking constructor on attached `DOMElement`, but the UAF bug can only manifest with specific initialization order. Note that CLEVEREST performs system-level testing, hence, no false positives in our result. We also confirmed that there is no flakiness.

However, for Poppler PDF parser, which requires a complex input format, CLEVEREST could only reproduce one bug. At least, it reached the changed code in half of the commits and exposed a difference in one commit. CLEVEREST consistently fails to generate an effective input in both scenarios for issues #1305 and #1381 of Poppler. While the generated input contains some related components required to test the commit, it does not satisfy the validity constraints required to successfully parse the input. For instance, CLEVEREST can generate an input that already contains the key elements necessary for revealing the bug in Issue #1305, including an annotation of type `Highlight` with an appearance stream and a `Resources` dictionary with an `ExtGState` entry. The failure to reach the commit occurs because Poppler strictly checks the existence of `QuadPoints` and `Rect` properties in the annotation object and exits early before reaching the changed code. If CLEVEREST knew these validity constraints and added these specific elements, as we confirmed, the generated PDF input could have reached the commit and even demonstrated an output difference.

CLEVEREST also struggles to generate bug-reproducing inputs for Z3 due to extremely simple commit messages that provide no information about the bug or fix. All four commits use the generic message "fix #[Issue ID]," with no further details. These are clear outliers (word count: 2) compared to the average commit message length in other programs (word count: 38).[4] This is another piece of evidence for our observation that the LLM can translate the *intention* of a well-described code change into a regression test case. We investigate this phenomenon further in **RQ3**.

Execution time and cost. CLEVEREST is well-suited for the time-constrained environment of a CI/CD pipeline. Execution times reported below include the full pipeline (commit extraction, test generation, and coverage collection). On average, it takes mostly less than one minute to generate XML, JavaScript, JSON, and Python inputs for Mujs, Libxml2, JerryScript, JQ, and MicroPython (~0.2$ on OpenAI) and still less than five minutes for PDF, PHP, and SMTLIB2 inputs, which involve different structural requirements (~0.5$).

Utility of CLEVEREST-generated test cases. We evaluate how developers can use the CLEVEREST-generated input in the pursuit of commit testing. Based on our execution analysis result, we found that even when CLEVEREST-generated inputs do not directly trigger the bug, the commit-reaching or output-changing inputs are still useful for the human developer to understand the program behavior and to guide the bug-finding process. CLEVEREST-generated test cases for BICs often "prepare" the precondition for the input to test the commit. For instance, the commit introducing Issue #550 of Libxml2 refines the XML parser's logic for checking the occurrence of the '<' character in entities. CLEVEREST catches the *"intention" of the commit* and generates an input containing a '<' character for the entity (e.g., CLEVEREST generated `<!ENTITY test "<notallowed>">`)

[4]Full commit message length statistics are available in Supplementary Material [17].

to reach the commit. The bug appears with the command `xmllint --dropdtd`, which removes the Document Type Definition (DTD) section from the document, which includes the entity definitions; thus, if Cleverest-generated input has an entity reference, which gets removed by the command (e.g., `<!DOCTYPE[<!ENTITY test>]<o><t test="&test;">...`), the bug would have been exposed.

Cleverest-generated test cases for BFCs can often be modified to trigger the bug. For instance, the commit that fixes Issue #141 of Mujs adds missing end-of-string checks in the regexp lexer for special syntax-indicating starting characters, including \x, \u, and \c. Cleverest generates the input `var regex = /\x/;` which exactly reflects the changed feature and produces a difference in program output from 'SyntaxError: invalid escape sequence' to 'SyntaxError: unterminated escape sequence,' demonstrating a newly added error-handling sequence in the commit. To trigger the bug in the program before the BFC, the regex has to have a very long string that starts with one of those special characters—information that cannot be derived from the commit information alone. Yet, the human developer can easily modify the Cleverest-generated input to contain a long string that reproduces the bug that was fixed by that commit. We further investigate the utility of Cleverest-generated test cases in **RQ4** as fuzzing seeds.

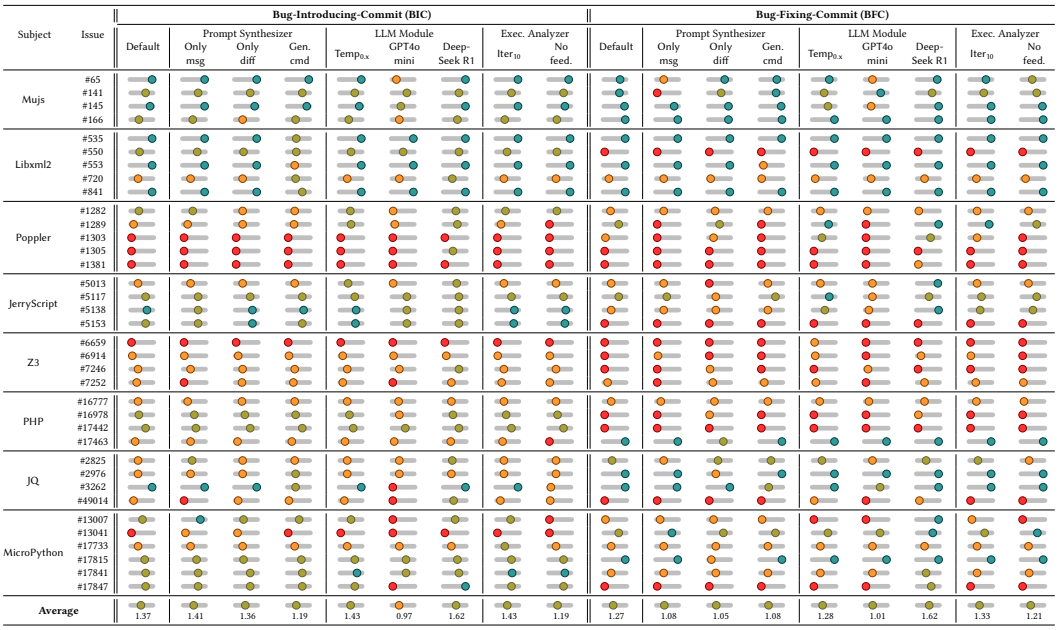
RQ1. Cleverest performs well for the commits to programs that take more human-interpretable formats but struggled to generate the right structure for the more complex format. With a short execution time of minutes, Cleverest is well-suited to be used in the CI/CD pipeline. As they are easy to read and may already be halfway there, Cleverest-generated inputs can also be used as a starting point for manual regression testing, even if they do not trigger any bugs in the given commit.

RQ2. Ablation Study

Table 3 presents the results of an ablation study on how Cleverest performs under various configurations and hyperparameter values for the bug-finding and bug-reproduction scenarios. Specifically, we evaluate how well Cleverest performs compared to the default configuration if we only used the commit message but not the diff or only used the commit diff but not the message (*commit information*), let it generate the command line prompt in addition to the test input (*task difficulty*), used `GPT-4o mini`/`DeepSeek-R1` as less/more powerful LLMs or maximized the LLM temperature (*LLM module*), used ten (10) iterations instead of five (5) in the feedback loop (*number of iterations*), or dropping the execution analysis result from the feedback (*feedback utility*).

Commit Information. We evaluate the impact of only using the commit message but not the diff (Only msg) or only using the commit diff but not the message (Only diff) and find that the results reproduce as long as the *intention* of the change is captured. In the bug-finding scenario, providing only the commit message or diff only changes the score slightly from 1.37 (meaning 'sometimes output-changing') to 1.41 and 1.36, respectively. However, in the bug-reproduction scenario, we observe cases where a single source is insufficient: some commit messages lack detail while others are too high-level compared to the diff. Specifically, some messages are too brief (e.g., "Fix #IssueId") and provide less information than the code changes (e.g., Issue #141 in Mujs, where the diff details specific escape sequences absent from its message). In contrast, other messages provide critical high-level intent (e.g., MicroPython Issue #13087 "Fix int.to_bytes() buffer size check") that goes beyond the literal syntax changes in the diff, from which the intent can be too difficult to infer. This discrepancy between the commit message and code diff makes having both sources of information meaningful, and removing the diff (Only msg) or the message (Only diff) causes a performance drop from 1.27 to 1.08 and 1.05 (meaning 'at least reaching'), respectively.

Table 3. **RQ2** results. Average effectiveness score on a slider representing effectiveness score ∈ [0, 3]



We seek deeper case studies on the informativeness of commit messages and its impact on test generation in **RQ3**.

Task Difficulty. We evaluate the impact of having CLEVEREST generate the command-line prompt in addition to the regression test case (Gen. cmd) for Libxml2, Poppler, and JQ, which require a specific command-line prompt to find or reproduce the bug. Our results show a noticeable but limited negative effect on CLEVEREST's performance for 7 out of 28 (25%) commits, changing the overall average score from 1.37 to 1.19 and 1.27 to 1.08 for the bug-finding and bug-reproduction scenarios, respectively. This is primarily because some programs have vast input space for command line prompts, while the bugs can only be triggered by specific values. For example, JQ #3262 was triggered by command `jq .[54E100]=7` in the original PoC. CLEVEREST could reliably generate commands with the array index feature to reach code changes, but only reproduced the bug in 2 out of 10 repetitions. This suggests that the LLM is capable of generating relevant command-line prompts for regression testing, though it struggles when the input space is vast.

Interestingly, CLEVEREST found one otherwise *undiscovered bug* when asked to also generate the command line prompt for reproducing Issue #550 of Libxml2 2. This bug was unrelated to Issue #550 but the bug-triggering command line `--xpath --dropdtd` was related to the *same feature* that was changed. This confirms our intuition that CLEVEREST generates regression test cases from the intention more than the actual code changes.

LLM Module. We evaluate the impact of LLM size on CLEVEREST's performance and find that model power significantly affects effectiveness. Using a weaker LLM (`GPT-4o mini`) reduces effectiveness for 16 issues in the bug-finding scenario, with test cases failing to reach code changes in 5 BICs. The average score drops from 1.37 to 0.97. For the bug reproduction scenario, effectiveness was reduced for 12 patches. In 4 cases, test cases fail to reach code changes, and the average score drops from 1.27 to 1.01. Conversely, a stronger LLM (`DeepSeek-R1`) enables CLEVEREST to find more bugs. In the bug-finding scenario, the number of bugs increased from 7 to 10, with 1 fewer in

JerryScript, but 4 additional bugs in JQ, Poppler, PHP, MicroPython, respectively, while `GPT-4o` failed to find any for the last three subjects. In the bug-reproduction scenario, the number of detected bugs increases from 13 to 20, with 1 fewer in Mujs, but 2 additional bugs in JerryScript, 2 in JQ, and 4 in MicroPython. The average score rises from 1.37 to 1.62 in the bug-finding scenario and from 1.27 to 1.62 in the bug-reproduction scenario. These results indicate that effectiveness heavily depends on the LLM's size/capacity, particularly for generating complex inputs.

We also evaluate an increase in temperature (i.e., the degree of confabulation). The score increased from 1.37 to 1.43 for bug-finding and 1.27 to 1.28 for bug-reproduction. This implies that maximizing temperature would also lead to an increase in test case diversity.

Number of feedback iterations. We evaluate the impact on effectiveness if we increase the number of feedback iterations from five (5) to ten (10) and find small positive impact for bug-finding (from 1.37 to 1.43) and bug-reproduction (from 1.27 to 1.33). In every iteration, Cleverest appends the execution feedback of the previous iteration from the Execution Analyser to the synthesized prompt (Fig. 2). After increasing the number of iterations, one case (1) turned from failing to reaching; two cases (2) turned from reaching to bug-triggering; one case (1) turned from output-changing to bug-triggering; Overall, we see the greatest change into the bug-triggering category when number of iterations is increased, at the cost of execution time.

Feedback utility. We evaluate the impact of dropping the execution analysis result from the feedback, i.e., having only the record of previously generated inputs in the prompt, and find that it has a negative impact on the effectiveness of Cleverest. In the bug-finding scenario (from 1.37 to 1.19), the decrease is obvious for ⟨Poppler, #1289⟩, ⟨PHP, #17463⟩ , and ⟨MicroPython, #13007⟩, where the generated test case now even fails to reach the code changes. In the bug-reproduction scenario (from 1.27 to 1.21), Cleverest's effectiveness reduces especially for ⟨Poppler, #1303⟩ and ⟨MicroPython, #13007⟩ where the generated test case now fails to reach the code changes. The result indicates that the execution feedback is crucial for generating the regression test input.

Taking a closer look, we found that the execution feedback was especially helpful to the LLM by pointing out invalid components in the input. For instance, the test case generated for Mujs in the first iteration often contains token `console`, which is unsupported by the program and causes error: `ReferenceError: 'console' is not defined`. The feedback enables the LLM to refine its inputs, progressively avoiding invalid components until it finally generates a valid, bug-triggering input.

RQ2. We found that Cleverest continues to generate effective test cases that find bugs in the changed feature as long as the intention is obvious which feature is changed, i.e., even when only an expressive commit message is provided. We also found that asking Cleverest to generate the command line prompt itself had only a limited negative effect on its performance; it even found a bug that was related to the changed feature but was only fixed much later. Reducing the model size or dropping the execution feedback both reduced the effectiveness. Doubling the number of iterations (at double the cost) increased the effectiveness of Cleverest.

RQ3. Effectiveness of Commit Messages

To investigate how informative and self-contained commit messages need to be for generating regression test cases without access to the code, we enhance or reduce the information in commit messages and observe how Cleverest's performance changes under these modifications, compared to using the original commit messages alone. In this study, we focus exclusively on bug reproduction because the intention behind a bug-fixing change is straightforward, while bug-inducing commits often conflate multiple intentions (e.g., adding a feature and refactoring), which makes them less suitable for controlled evaluation.

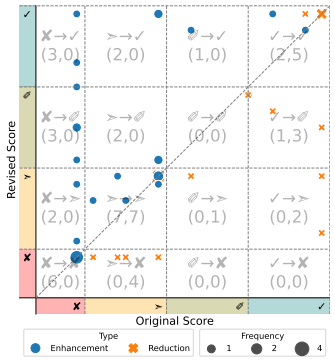


Fig. 3. Changes in bug reproduction scores from original (x-axis) to modified (y-axis) commit messages. ● and ✖ mark enhancements and reductions (higher is better). The background label (X, Y) means that X and Y scores moved here after enhancement and reduction, respectively.

(Subject, Issue)	Message	Ver.	Generated Input	Result
(Z3, #7252)	fix #7252 *segmentation fault in nested re.loop formula processing when handling specific boundary values for nested re.loop expressions with same bounds of 0 and 1*	Orig.	`(declare-fun a () Int) (assert (= a (+ a 1))) (check-sat)`	✘
		Enhc.	`(declare-fun a () (RegEx String)) (assert (str.in.re "a" ( re.loop (re.loop a 0 1) 0 1))) (check-sat)`	✓
(Mujs, #65)	Fix issue #65: Uninitialized name *and length properties* in Function.prototype *causing a null pointer dereference when 'Function.prototype' is converted to a string*	Orig.	`Function.prototype.call.name;`	➢
		Enhc.	`String(Function.prototype);`	✓
(JerryScript, #5153)	Fix parsing of invalid asterisk symbol after private field symbol *, which previously caused a segmentation fault when an asterisk was used after an async private method declaration.*	Orig.	`class A { #private* }`	✘
		Enhc.	`class C { async #*foo() {} }`	✓
(JerryScript, #5117)	Fix arguments object in **eval-ed** functions in static class initializers	Orig.	`class MyClass { static { eval("function testFunc() { return arguments; }"); console.log(testFunc(1, 2, 3));}}`	✓
		Redu.	`class MyClass { static { function testFunc() { console.log(arguments); } testFunc(1, 2, 3);}}`	➢

Table 4. Examples of how intention in commit message affects test case generation. (Enhancement: *teal & italic*, Reduction: **red & bold**)

Specifically, we conduct the following procedure:

(1) We first examine each of the 36 BFC and its corresponding bug report to identify the key elements describing the bug and its resolution.

(2) *Commit Message Reduction.* For BFCs where CLEVEREST was able to generate test cases that at least reach the modified code (i.e., 22 BFCs), we manually remove some of the identified key elements from the commit message, leaving a reduced version with less critical information. For a representative example, see last row in Table 4.

(3) *Commit Message Enhancement.* There are 7 BFCs where CLEVEREST always generates a bug-triggering test case. For all other BFCs (i.e., 29 BFCs), we enhance the commit message. To standardize the enhancement process across commits, we employ another LLM (`gemini-2.5-flash`) which is given the original commit and the corresponding bug report. We then manually verify that the enhanced message includes the key elements identified in Step 1. The details of the LLM configuration and prompts are provided in the Supplementary Material [17]. For representative examples of such enhancements, see top three rows in Table 4.

Figure 3 summarizes the changes in CLEVEREST's effectiveness when using modified commit messages compared to the original ones. The x-axis and y-axis denote effectiveness scores with the original and modified commit messages, respectively. Circles and crosses mark enhancements and reductions, with marker size showing commit counts. Markers above the diagonal represent improvements, those below represent decreases, and the diagonal line indicates no change.

The results show that enhancing commit messages generally improves CLEVEREST's effectiveness. This suggests that well-detailed commit messages substantially help the LLM capture the intention of the change and generate more effective test cases. After enhancement, 59% (17/29) of the commits exhibited improved bug reproduction scores. Notably, among 14 commits where CLEVEREST initially failed to generate test cases that reached the modified code, 5 were able to trigger the bug, while 2 and 1 managed to change the output and reach the modified code, respectively, after enhancement. Combining these with the cases where the original commit message already triggered the bug, CLEVEREST successfully generated bug-triggering tests in 18 out of 36 cases, given only the commit message-level natural language description of the code change.

To gain insights into the results, we manually examined the quality of the original commit messages to understand why they sometimes failed to guide regression test generation, and how the enhanced messages improved the outcome.

In failing cases (either ✘ or ➢), we found that some original messages contained almost no useful information (e.g. Z3's "Fix #IssueId"), leaving CLEVEREST with little basis for generating test cases that could even reach the modified code. Others described only the root cause in the implementation without mentioning the bug's manifestation (e.g., (Mujs, #65)'s "Fix: Uninitialized name in Function.prototype function"), which made it difficult for CLEVEREST to infer the developer's intention and produce effective tests. Most commit messages did reference the fix, but often lacked precision in describing the conditions under which the bug occurred (e.g. (JerryScript, #5155)'s "Fix parsing of invalid asterix symbol after private field symbol"). After enhancement—when the commit messages included details about the bug's nature and manifestation (Figure 4)—CLEVEREST was able to infer the bug-triggering context and generate effective bug-revealing test cases.

We observed only two cases where the score dropped slightly (from 1.0 to 0.9 and 2.1 to 2.0, respectively). On manual inspection, we found that, after enhancement, the message led CLEVEREST to generate malformed PDF inputs that, in one of ten runs, triggered parser syntax errors before reaching the modified code. This drop stems from execution randomness rather than a weakness of the enhancement.

Conversely, reducing commit messages tends to decrease CLEVEREST's effectiveness. This not only reinforces our claim but also confirms that the finding of **RQ2**—that commit messages alone are sufficient to generate effective test cases—is not coincidental or merely a result of LLM memorization. After reduction, 50% (11/22) of the commits exhibited decreased bug reproduction scores, with a median of only two words reduced. Notably, among 13 commits where CLEVEREST initially generated test cases that triggered bug, 1 could not even reach modified code, and 4 were only able to reach the code. Among 9 commits where CLEVEREST initially generated test cases that could reach the changed code, 3 were not even able to reach the code after reduction. For example, by removing just one word "eval-ed" from the commit message of JerryScript #5117, the generated tests can only reach the changed code but never trigger the bug.

RQ3. Our results demonstrate that CLEVEREST's ability to generate effective regression test cases from commit messages alone is not coincidental: in 11/22 cases, the effectiveness score dropped when key information about the bug was removed from the commit message. Furthermore, when sufficient detail is present, CLEVEREST can produce effective regression test cases. With enhanced commit messages, 18/36 cases were able to trigger the bug using only the natural language description in the commit message. This result highlights that LLMs are capable of generating regression tests solely from natural language descriptions of code changes.

RQ4. Comparison to the State-of-the-Art

In this section, we compare CLEVEREST against two baselines: primarily WAFLGo, the state-of-the-art in regression test generation, and existing LLM-based test generation approaches [37]. Since prior LLM-based techniques are not designed for directed, regression-focused testing, we instead compare against unguided LLM-based test generation with no commit information used. We briefly report this comparison at the end of the section, with full details in the Supplementary Material [17].

For WAFLGo, we found that it does not work for any commits of Z3 and PHP due to its static analysis module exceeding our server's 251GB RAM. It also failed to build for commits `b7e3bae` and `de51531` of JerryScript and commits `e702046`, `f397a3e`, `c0252d7`, and `4b013ec` of MicroPython.

Table 5. Results for bug finding and bug reproduction effectiveness of CLEVEREST, CLEVEREST + fuzzing (CLEVFUZZ), and WAFLGo across ten repetitions on the 50 subjects where WAFLGo was runnable. For each configuration, we report initial seed effectiveness (✘ to ✓), number of WAFLGo campaigns that found the bug and time-to-exposure (*T.O.* = timeout after 24 hours), CLEVEREST's average effectiveness score (slider), number of commit-reaching / output-changing / bug-triggering repetitions, and time spent. '-/-' indicates CLEVFUZZ was skipped as CLEVEREST already caught the bug in 10/10 repetitions. [Bug_{all}] is union of them.

	Subject (# seeds)	Issue	WAFLGo Init	WAFLGo Bug	WAFLGo T (h:m:s)	CLEVEREST ✘➢🩹✓	CLEVEREST Reach	CLEVEREST Change	CLEVEREST Bug	CLEVEREST T (h:m:s)	CLEVFUZZ Bug	CLEVFUZZ T (h:m:s)	CLEVFUZZ Bug_{all}
Bug-finding	Mujs (19)	#65	✘	10/10	04:43:56		10/10	10/10	10/10	00:00:07	-/-	-	10/10
		#141	✘	0/10	T.O.		10/10	10/10	0/10	00:00:45	10/10	03:25:05	10/10
		#145	✘	10/10	00:17:21		10/10	8/10	8/10	00:00:37	2/2	00:00:00	10/10
		#166	✓	10/10	T.O.		10/10	1/10	0/10	00:00:57	0/10	T.O.	0/10
	Libxml2 (14)	#535	✓	10/10	T.O.		10/10	10/10	10/10	00:00:05	-/-	-	10/10
		#550	🩹	0/10	T.O.		10/10	2/10	0/10	00:00:27	0/10	T.O.	0/10
		#553	✓	10/10	T.O.		10/10	10/10	10/10	00:00:06	-/-	-	10/10
		#720	🩹	0/10	T.O.		10/10	0/10	0/10	00:00:53	4/10	18:08:01	4/10
		#841	✓	10/10	T.O.		10/10	10/10	10/10	00:00:05	-/-	-	10/10
	Poppler (100)	#1282	➢	6/10	00:06:23		8/10	3/10	0/10	00:03:16	8/8	00:00:10	8/10
		#1289	➢	0/10	T.O.		4/10	0/10	0/10	00:02:45	3/4	06:00:04	3/10
		#1303	✘	0/10	T.O.		0/10	0/10	0/10	00:03:38	-/-	-	0/10
		#1305	✘	0/10	T.O.		0/10	0/10	0/10	00:04:27	-/-	-	0/10
		#1381	➢	0/10	T.O.		0/10	0/10	0/10	00:03:22	-/-	-	0/10
	JerryScript (19)	#5117	➢	4/10	00:13:02		10/10	10/10	0/10	00:01:11	10/10	00:01:41	10/10
		#5138	➢	0/10	T.O.		10/10	10/10	2/10	00:01:18	8/8	00:00:37	10/10
		#5153	➢	0/10	T.O.		10/10	10/10	0/10	00:01:35	10/10	00:00:59	10/10
	JQ (38)	#2825	➢	0/10	T.O.		10/10	0/10	0/10	00:00:23	10/10	04:42:34	10/10
		#2976	➢	2/10	01:55:03		10/10	0/10	0/10	00:00:22	10/10	01:21:39	10/10
		#3262	✓	10/10	T.O.		10/10	10/10	10/10	00:00:09	-/-	-	10/10
		#49014	✘	0/10	T.O.		2/10	2/10	0/10	00:00:32	2/2	07:51:24	2/10
	MicroPython (13)	#13007	➢	0/10	T.O.		9/10	7/10	0/10	00:00:27	7/9	07:07:12	7/10
		#13041	✘	0/10	T.O.		0/10	0/10	0/10	00:00:31	-/-	-	0/10
		#17733	➢	0/10	T.O.		10/10	0/10	0/10	00:00:26	6/10	12:35:08	6/10
		#17815	➢	0/10	T.O.		10/10	9/10	0/10	00:00:26	4/10	17:54:34	4/10
		#17847	✘	0/10	T.O.		10/10	10/10	0/10	00:00:40	2/10	20:33:41	2/10
	Aggregate			**10/26**	**19:39:50**		**22/26**	**17/26**	**7/26**	**00:01:08**		**05:42:01**	**20/26**
Bug-reproduction	Mujs (19)	#65	➢	0/10	T.O.		10/10	6/10	6/10	00:00:19	1/4	18:03:42	7/10
		#141	✘	5/10	00:12:47		10/10	10/10	1/10	00:00:31	9/9	01:43:32	10/10
		#145	✘	10/10	00:08:26		10/10	9/10	9/10	00:00:28	1/1	00:00:03	10/10
		#166	✓	10/10	T.O.		10/10	10/10	10/10	00:00:07	-/-	-	10/10
	Libxml2 (14)	#535	✓	10/10	T.O.		10/10	10/10	10/10	00:00:05	-/-	-	10/10
		#550	✘	0/10	T.O.		0/10	0/10	0/10	00:00:29	-/-	-	0/10
		#553	✓	10/10	T.O.		10/10	10/10	10/10	00:00:13	-/-	-	10/10
		#720	🩹	0/10	T.O.		7/10	0/10	0/10	00:01:44	7/7	03:09:11	7/10
		#841	✓	10/10	T.O.		10/10	10/10	10/10	00:00:05	-/-	-	10/10
	Poppler (100)	#1282	➢	0/10	T.O.		9/10	0/10	0/10	00:02:03	1/9	22:09:30	1/10
		#1289	➢	0/10	T.O.		8/10	6/10	6/10	00:01:30	2/2	00:00:08	8/10
		#1303	✘	0/10	T.O.		1/10	1/10	0/10	00:02:12	0/1	T.O.	0/10
		#1305	➢	0/10	T.O.		0/10	0/10	0/10	00:02:27	-/-	-	0/10
		#1381	➢	0/10	T.O.		0/10	0/10	0/10	00:02:27	-/-	-	0/10
	JerryScript (19)	#5117	➢	0/10	T.O.		10/10	5/10	5/10	00:00:50	2/5	20:54:43	7/10
		#5138	✘	0/10	T.O.		10/10	0/10	0/10	00:01:14	4/10	17:50:21	4/10
		#5013	✘	0/10	T.O.		10/10	0/10	0/10	00:01:12	10/10	03:02:42	10/10
	JQ (38)	#2825	➢	0/10	T.O.		10/10	1/10	0/10	00:00:28	2/5	19:20:31	2/10
		#2976	➢	1/10	19:03:38		10/10	10/10	10/10	00:00:03	-/-	-	10/10
		#3262	✓	10/10	T.O.		10/10	10/10	10/10	00:00:08	-/-	-	10/10
		#49014	✘	0/10	T.O.		0/10	0/10	0/10	00:00:32	-/-	-	0/10
	MicroPython (13)	#13041	🩹	0/10	T.O.		10/10	10/10	0/10	00:00:21	10/10	02:58:33	10/10
		#17733	🩹	0/10	T.O.		10/10	0/10	0/10	00:00:31	5/10	15:00:18	5/10
		#17847	🩹	0/10	T.O.		0/10	0/10	0/10	00:00:29	-/-	-	0/10
	Aggregate			**8/24**	**21:48:32**		**19/24**	**14/24**	**11/24**	**00:00:51**		**06:11:24**	**18/24**

Hence, we have 50 commits for the comparison. Finally, for WAFLGo, we report only the fuzzing campaign time, as done in prior work, and do not include the static analysis pre-processing overhead required to construct its directed fuzzing guidance—a choice that favors WAFLGo in our comparison.

RQ4.1 Comparison with WAFLGo

Table 5 contains the results for the comparison. We first evaluate both tools head to head and then explore the opportunities to fuzz interesting Cleverest-generated test cases (ClevFuzz).

Effectiveness. In the bug reproduction scenario, Cleverest (without initial seeds) outperforms WAFLGo (which requires seeds): it reproduces bugs in 11 of 24 BFCs, compared to WAFLGo's 8 of 24. This is a notable improvement, especially given that Cleverest operates without a seed corpus. In the bug finding scenario, Cleverest reaches the modified code in 22 of 26 BICs and changes the output in 17 of them, both higher than WAFLGo. However, it triggers the actual bug in 7 of 26 BICs, fewer than WAFLGo (10 of 26). This result is expected, since Cleverest relies on the intention expressed in commit messages (as shown in **RQ3**), and BICs often describe new features or improvements rather than the bug itself. We further examine WAFLGo's dependence on the initial seed corpus in the next section.

Execution time. Cleverest is substantially faster than WAFLGo, making our LLM-based approach more suitable as part of the CI/CD pipeline, which runs under strict time and resource constraints. A typical fuzzing campaign is set to 24 hours. Apart from ten cases where the initial seeds already exposed the bug introduced or patched by a commit, WAFLGo takes between five minutes and five hours on average to find the bugs, and otherwise times out. For the commit fixing Issue #2976 of JQ, WAFLGo took more than 19 hours. Cleverest's execution time is bounded by the number of feedback iterations (fixed to 5 in our experiments) and requires roughly a minute for Mujs, Libxml2, JerryScript, JQ, and MicroPython and mostly less than three minutes for Poppler, Z3, and PHP.

Fuzzing Cleverest's seeds for improved effectiveness. Since fuzzing works well if seeds are close already, why not try using the Cleverest-generated inputs as fuzzer seeds (ClevFuzz)? During qualitative analysis of change-reaching and output-changing Cleverest test cases for **RQ1** (§3), we found that they are often not "very far" from bug-revealing; a few modification can turn them to bug-finding test cases. This idea is further inspired by LLMs being used as seed generators for fuzzing, a successful strategy in the 2024 DARPA AI Cyberchallenge (AIxCC).[5]

For each trial with a change-reaching or output-changing Cleverest-generated input, we run a 24-hour AFL++ v4.21c campaign with the default configuration, using that test case as the only seed. The denominators in Column [ClevFuzz. Bug] of Table 5 indicate the number of such trials among 50 subjects. Specifically, 143 Cleverest-generated test cases reached the changes but failed to reveal the bug in 17 BICs, while 83 reached the change but did not reveal the bug in 13 BFCs. The full results across all 72 commits are provided in Supplementary Material [17].

Results. As shown in the last column of Table 5, our LLM-seeded fuzzer ClevFuzz outperforms WAFLGo, which is started on a user-provided seed corpus. Overall, ClevFuzz finds bugs in 20 out of 26 BICs and 18 out of 24 BFCs, significantly outperforming WAFLGo (10 and 8), finding more than twice as many bugs in both scenarios.

Specifically, ClevFuzz can find 13 additional bugs for BICs and 7 for BFCs after the fuzzing campaign. The fuzzing turned 96 out of 143 (67%) Cleverest-generated test cases for BICs and 54 out of 83 (65%) for BFCs into bug-revealing test cases. This indicates that Cleverest-generated test

[5]Trail of Bits, one of the finalists says: "Our system uses large language models (LLMs) to generate seed inputs for fuzzing, significantly reducing the time needed to discover vulnerabilities. This innovative approach helps us work within the competition's strict time constraints." [1]

cases are often close to bug-revealing already, and fuzzing can effectively explore the neighborhood of these test cases to find the bugs.

Timewise, this LLM-based zero-shot fuzzing approach remains more than twice as fast as WAFLGo. Considering the combined time for CLEVEREST and fuzzing, CLEVFUZZ takes roughly 6 hours[6] for the subjects where WAFLGo is applicable, while WAFLGo takes roughly 20 and 22 hours for bug finding and bug reproduction scenarios, respectively. Indeed, this is an intriguing result as WAFLGo can be considered few-shot (i.e., starting from a user-provided corpus) while our approach is zero-shot (i.e., no examples needed).

RQ4.2 CLEVEREST as Zero-Shot Regression Test Generator

From the perspective of CLEVEREST as a zero-shot regression test generator, we wanted to explore the dependence of WAFLGo's effectiveness on the initial set of seed files (i.e., valid XML, JS, PDF, JSON, and Python files). The authors of WAFLGo [38] used a sound and fair seed corpus selection strategy for their experiments. We follow its setting to use initial seeds from the UNIFUZZ benchmark [16], except for MicroPython where we pick seeds from its builtin test suite as UNIFUZZ benchmark does not contain Python seeds. For Mujs, Libxml2, Poppler, JerryScript, JQ, and MicroPython there are 19, 14, 100, 19, 38 and 13 initial seeds for WAFLGo. Nevertheless, in ten cases, the initial corpus already reveals Issues #166 of Mujs, #535, #553, and #841 of Libxml2, and #3262 of JQ introduced or fixed in the corresponding commits (cf. Column *WAFLGo.Init* in Tab. 5). This raises the question of how close to triggering the corresponding bugs are WAFLGo initial seeds.

```
 (function() {
 var a = 1, b = 2;
 for(var i = 0; i < 1e5; i++) { /a/g
- if(a === b) {
+ if(a =Error== b) {
    throw new Error;
  }
 })();
```

Listing 1. Mujs #65

```
 c = 30000;
 a = [];
 for (i = 0; i < 2 * c; i += 1)
   a.push(i%c);
 a.sort(function (x, y) {
   return x - y; });
-print(a[2 * c - 2]);
+print(a[2 * a - 2]);
```

Listing 2. Mujs #145

Results. Table 5 (Col. *WAFLGo.Init*) shows the effectiveness of the initial corpus in terms of reaching the code changes (➢) and revealing changes in the output (✎). Among the 50 cases, in addition to the ten (10) bug-revealing cases, in 25 cases, the initial seed corpus at least already reaches the code changes. In two (2) additional cases, only a few characters need to be changed to reveal the bug. Listings 1 and 2 above show such examples for Issues #65 and #145 of Mujs. Given this result, we find that CLEVEREST, as a zero-shot regression test generator, also makes an excellent seed generator for regression greybox fuzzers, like WAFLGo.

RQ4. While CLEVEREST is substantially faster than WAFLGo, CLEVEREST performs better than WAFLGo in bug reproduction and slightly worse in bug finding. However, by upgrading CLEVEREST by fuzzing the generated test cases, our zero-shot approach outperforms WAFLGo, whose performance depends on a user-provided seed corpus, which, as we found, happens to be quite close to bug-revealing already. Hence, we find that CLEVEREST, as a zero-shot regression test generator, also makes an excellent seed generator for regression greybox fuzzers.

Finally, we elaborate the results versus *undirected*-CLEVEREST, which excludes commit information. We find that the undirected variant significantly under-performs our proposed directed variant (i.e., 4 bug versus 20 bugs). The effectiveness score drops from 1.32 (sometimes output-changing) to

[6]Including the time-out of the fuzzing campaign, which is 24 hours.

0.63 (often not reaching the changes), and the average execution time increases from 1 minute to ~3 minutes. These results highlight the importance of commit information for LLM-based regression test generation. Detailed experimental results are provided in Supplementary Material [17].

4 Threats to Validity

Construct Validity. A key concern is the potential for data leakage and memorization by LLMs, where test set information may inadvertently influence training. In this study, the risk pertains to LLMs memorizing the regression test cases from the WAFLGo benchmark (cf. Table 1). To assess this risk, we computed the similarity, in terms of Levenshtein ratio [3], between the Cleverest-generated test cases and the available bug-triggering test cases that were provided along with the bug report (not the commit). We find that the majority of Cleverest-generated test cases are less than 7% similar (mean 10%, max. 40%) to the ones available online. A significant difference suggests that the LLM did not simply memorize them. We also note that in an experiment where Cleverest was asked to generate the command line prompt, as well, a bug was found that—while unrelated to the feature changed in the targeted bug fix—was only fixed after LLM's cut-off date (cf. **RQ2**). Our experiment results in **RQ3** with modified commit message also proves that Cleverest does not simply memorize the source code or commit history, as the performance highly depends on the intention. Removing a single word can lead to a significant performance drop.

Internal Validity. We identified three main threats to internal validity: implementation correctness, confabulation of the LLM, and the randomness of our results. To ensure correctness, we conducted thorough code reviews and testing. Our replication package is made available for transparency. We used publicly accessible APIs of `GPT-4o`, `GPT-4o mini`, and `DeepSeek-R1`, adhering strictly to their documented usage guidelines. We mitigate the impact of confabulation; all generated test cases are actually executed on subject programs to validate the test outcome. To handle the randomness in our results, we repeated each experiment within the available budget, i.e., ten (10) times, and reported the findings across all trials.

External Validity. We do not claim the generality of our results but consider our experiments as an important case study for six open-source C programs that take highly structured, human-readable inputs. We sought to test the capabilities of an LLM as a regression test generation tool and carefully established benchmark selection criteria to align with this goal. We selected *all* programs from the WAFLGo benchmark that apply and *all* of their commits. The findings may not extend to programs with less structured or non-human-readable input formats. Finally, our approach does not aim to replace assertion-based regression test suites for well-specified functionality. Instead, it complements them in domains where expected behavior is underspecified or difficult to encode, and where regressions commonly manifest as crashes or semantic inconsistencies.

Bug Types and Detection Mechanisms. Our approach is agnostic to the type of bug being detected and does not rely on memory-specific properties. The only assumption is the availability of an automated mechanism that flags failures when executing generated test cases. In our evaluation, we instantiate this mechanism using ASan to detect memory-related bugs. However, our approach already detects non-memory-related failures in the benchmark. For example, bugs #5117/5138/5153 in JerryScript are detected via internal assertion failures (ERR_FAILED_INTERNAL_ASSERTION) rather than memory violations. Other detectors, such as developer-provided assertions, runtime checks, differential testing oracles, or semantic consistency checks, could be integrated without changes to the test generation pipeline.

5 Related Work

LLMs for Automatic Unit Test Generation. The field of automated test case generation has evolved significantly, particularly with the advent of deep learning techniques [5, 32, 35].

More recent advancements have explored the adoption of LLMs for automatic unit test generation. TESTPILOT [28] relies on LLMs to generate unit tests by providing the signature, implementation and usage documentation of the function under test, and iteratively refine the tests with execution feedback. ChatUnitTest [6] overcomes the prompt context window limit by adaptively selecting the most relevant code elements, and mitigates errors in tests through a generation-validation-repair loop. SymPrompt [26] generates test inputs for executing a specific path identified during the static analysis. HITS [34] tackles the problem of testing complex methods by using LLMs to decompose method-to-test and generate unit tests slice by slice. COVERUP [23] achieves high coverage by including coverage information in feedback to generate unit tests targeting uncovered code. These works share the same goal of generating better unit tests to improve overall coverage, while Cleverest aims to generate system-level directed input to test specific regression changes. More generally, we are interested in regression test generation as a machine translation task from the intention of a change to a test of that change.

LIBRO [15] relies on LLMs to generate and rank candidate test cases from bug reports in structured benchmarks like Defects4J [14]. Testora [24] uses LLMs to identify behavioral changes in regression tests generated for API calls from four Python libraries. Meta also internally deploys LLM-based test generation system [2, 12] to find software regression using mutation testing. This system assumes the presence of mutants and leverages them to strengthen regression test suites, while Cleverest investigates whether LLMs can independently produce such precise tests in the first place.

Cleverest is different from previous works in the following ways:

- It generates system-level input, which can be user-provided and external to the actual code implementation. We demonstrated that LLM can generate effective test cases with lightweight *grey-box* information like an expressive commit message and execution feedback. In contrast, previous works generate unit tests that depends on a specific codebase, requiring heavy *white-box* analysis about the target method (e.g., API signature, full source code);
- It is the first evaluation of LLMs with respect to regression test generation. In particular, Cleverest focuses on generating bug-revealing test cases from commits while previous works have addressed the *undirected* and the *unit test* generation problem to improve test suite's coverage;
- It conducts the first controlled experiments to study the impact of the developer's intention for a change in the form of *natural language* instead of specification on LLM-based test generation, and demonstrates that more specific expressions of intention increases an LLMs effectiveness.

Recent works [11, 25] have also highlighted the role of intention in other software engineering tasks. Guo et al. [11] propose a framework for code refinement, where review comments are translated into structured intentions and then used to guide LLMs in generating code modifications. Similarly, Qi et al. [25] formalizes validation intentions (e.g., objectives, preconditions, and expected results) to generate unit tests. Such approaches focus on extracting and encoding intentions to drive refinement or generation. In contrast, Cleverest does not formalize intention, but rather studies how intention expressed in commit messages affects the quality of generated test inputs.

LLMs for Fuzzing. Several fuzzers leverage LLMs to enhance fuzzing techniques, but each operates with different goals and methods. ChatAFL [21] uses LLMs to construct grammars for protocol messages, mutate inputs, and generate sequences to improve fuzzing efficiency, distinguishing it from Cleverest, which is not a protocol fuzzer. PromptFuzz [20], meanwhile, utilizes LLMs to iteratively generate fuzz drivers that explore API functions, making it fundamentally different from Cleverest, which focuses on fuzzing through input generation rather than fuzz driver creation. Fuzz4All [37] employs a pure-LLM approach to generate and mutate code snippets testing various compiler features with user-provided documentation and example code. Jiang et al. [13] conducted the first systematic study comparing LLMs with constraint-based techniques for directed test input

generation, evaluating their ability to reach specific target branches. Cottontail [31] uses LLMs to extends concolic execution for systematically testing programs taking highly-structured inputs. TELPA [39] combines program analysis with LLMs to cover hard-to-cover branches by extracting real usage scenarios and resolving inter-procedural dependencies. Sapia and Böhme [27] introduce evaluate the utility to test a software system end-to-end to bridge the reachability gap. In contrast, Cleverest generates a few *regression* tests *directed* towards a given commit in a zero-shot manner.

Feedback-guided LLM test generation. Recent work has confirmed that LLMs require data from external analysis to identify what to test next. MuTAP [7] iteratively augments prompts with selected mutants, enabling the model to generate additional tests that exclude remaining mutants and improve mutation score. MUTGEN [33] extends this idea with a pipeline that relies on mutant analysis, test fixing, and regeneration to maximize fault-detection capability. Panta [10] follows the same feedback-directed paradigm, but relies on hybrid program analysis rather than mutation testing: it extracts independent execution paths from a control-flow graph and combines them with dynamic coverage to guide the LLM toward under-tested behaviors. Cleverest shares this approach of analysis-guided prompting, but for a different objective: MuTAP, MUTGEN, and Panta generate unit tests for a single program version with coverage or mutation as the primary goal, while Cleverest generates system-level inputs to validate a specific commit, using differential execution across versions as driving information.

6 Discussion

Throughout the study, we have shown the significant potential of LLMs in generating structured, human-readable, system-level inputs for testing code changes. Our results highlight that even *without* advanced techniques such as fine-tuning or retrieval-augmented generation (RAG), LLMs, when combined with prompting and execution feedback, are capable of producing high-quality test inputs. These inputs can reflect the semantic meaning inside commits. In some cases, they directly trigger the bugs, and in other cases, they could be "repaired" to trigger the bugs. So there is potential for LLMs in directed software testing, enabling automated testing workflows that are typically more challenging for conventional tools.

One notable observation is the LLM's ability to generate meaningful input without access to source code context, as long as we describe the intention with accurate natural language. For the philosophical minded, this finding resonates with Wittgenstein's statement in his Tractatus: "What can be shown cannot be said". Here what is said is the commit message describing the developer's intention and what is shown is the concrete executable test demonstrating the consequence of that intention. Our results show that the inverse might be true for automated software engineering: much of what can be "said" can indeed be "shown". Our work unlocks the potential of "vibe testing" as a new form of "language game" [36], where users describe high-level intention with natural language, and LLM-powered systems instantiate the meaning into concrete tests to trigger specific behavior of the underlying program, with this "vibe" becoming a tangible, testable reality.

While using our tool Cleverest standalone has already shown promising results in generating effective test cases in a short time, we believe the value of Cleverest shines even more in their subsequent use as a tool in a developer's hand. The LLM-generated test cases are easily comprehensible by human developers, as they are structured and human-readable; we demonstrated how human developers can modify these test cases to trigger bugs. They are also an excellent seed generator for regression greybox fuzzers; even a vanilla greybox fuzzer can perform similarly or better than WAFLGo, which requires a user-provided seed corpus.

7 Data Availability

The implementation code, data, and scripts used in this study are available at https://github.com/niMgnoeSeeL/cleverest.

Acknowledgments

We thank the anonymous reviewers for their constructive feedback and for helping us improve this paper. We thank Yi Xiang, the author of WAFLGo paper, for answering questions about running WAFLGo. This research is partially funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them. Besides funding this work, the funding source had no involvement in the conduct of the research and in the preparation of the article. This work is supported by ERC grant (Project AT_SCALE, 101179366). It is also partially funded by NextGenerationEU and by the 2023 STARS Grants@Unipd programme (Acronym and title of the project: "PatchThemAll: A Virtualization-Based Approach for Distributing Android Security Patches on Any Custom Android OS"). It is also partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972.

References

[1] The Trail of Bits Blog 2024. *Trail of Bits' Buttercup Heads to DARPA's AIxCC.* The Trail of Bits Blog. https://blog.trailofbits.com/2024/08/09/trail-of-bits-buttercup-heads-to-darpas-aixcc/

[2] Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. 2024. Automated Unit Test Improvement using Large Language Models at Meta *(FSE 2024)*. Association for Computing Machinery, New York, NY, USA, 185–196. doi:10.1145/3663529.3663839

[3] Max Bachmann. 2024. Levenshtein.ratio. https://rapidfuzz.github.io/Levenshtein/levenshtein.html#ratio.

[4] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2329–2344. doi:10.1145/3133956.3134020

[5] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. CodeT: Code Generation with Generated Tests. arXiv:2207.10397 [cs.CL] https://arxiv.org/abs/2207.10397

[6] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. 2024. ChatUniTest: A Framework for LLM-Based Test Generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering* (Porto de Galinhas, Brazil) *(FSE 2024)*. Association for Computing Machinery, New York, NY, USA, 572–576. doi:10.1145/3663529.3663801

[7] Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C. Desmarais. 2024. Effective test generation using pre-trained Large Language Models and mutation testing. *Inf. Softw. Technol.* 171, C (July 2024), 17 pages. doi:10.1016/j.infsof.2024.107468

[8] Zhengjie Du, Yuekang Li, Yang Liu, and Bing Mao. 2022. WindRanger: a directed greybox fuzzer driven by deviation basic blocks. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2440–2451. doi:10.1145/3510003.3510197

[9] Khashayar Etemadi, Bardia Mohammadi, Zhendong Su, and Martin Monperrus. 2024. Mokav: Execution-driven differential testing with llms. *arXiv preprint arXiv:2406.10375* (2024).

[10] Sijia Gu, Noor Nashid, and Ali Mesbah. 2025. LLM Test Generation via Iterative Hybrid Program Analysis. https://arXiv:2503.13580

[11] Qi Guo, Xiaofei Xie, Shangqing Liu, Ming Hu, Xiaohong Li, and Lei Bu. 2025. Intention is All you Need: Refining your Code from your Intention . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 1127–1139. doi:10.1109/ICSE55347.2025.00191

[12] Mark Harman, Jillian Ritchey, Inna Harper, Shubho Sengupta, Ke Mao, Abhishek Gulati, Christopher Foster, and Hervé Robert. 2025. Mutation-Guided LLM-based Test Generation at Meta. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering* (Clarion Hotel Trondheim, Trondheim, Norway) *(FSE Companion '25)*. Association for Computing Machinery, New York, NY, USA, 180–191. doi:10.1145/3696630.3728544

[13] Zongze Jiang, Ming Wen, Jialun Cao, Xuanhua Shi, and Hai Jin. 2024. Towards Understanding the Effectiveness of Large Language Models on Directed Test Input Generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 1408–1420. doi:10.1145/3691620.3695513

[14] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *ISSTA 2014, Proceedings of the 2014 International Symposium on Software Testing and Analysis*. San Jose, CA, USA, 437–440. Tool demo.

[15] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2312–2323. doi:10.1109/ICSE48619.2023.00194

[16] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, Kangjie Lu, and Ting Wang. 2021. UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2777–2794. https://www.usenix.org/conference/usenixsecurity21/presentation/li-yuwei

[17] Jing Liu, Seongmin Lee, Eleonora Losiouk, and Marcel Böhme. 2026. *Supplementary Material for "Evaluating LLM-Based Regression Test Generation"*. doi:10.5281/zenodo.19616583

[18] Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. 2024. Large Language Model-Based Agents for Software Engineering: A Survey. arXiv:2409.02977 [cs.SE] https://arxiv.org/abs/2409.02977

[19] Changhua Luo, Wei Meng, and Penghui Li. 2023. SelectFuzz: Efficient Directed Fuzzing with Selective Path Exploration. In *2023 IEEE Symposium on Security and Privacy (SP)*. 2693–2707. doi:10.1109/SP46215.2023.10179296

[20] Yunlong Lyu, Yuxuan Xie, Peng Chen, and Hao Chen. 2023. Prompt Fuzzing for Fuzz Driver Generation. arXiv:2312.17677 [cs.CR]

[21] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. 2024. Large Language Model guided Protocol Fuzzing. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*.

[22] Yaroslav Oliinyk, Michael Scott, Ryan Tsang, Chongzhou Fang, Houman Homayoun, et al. 2024. Fuzzing {BusyBox}: Leveraging {LLM} and Crash Reuse for Embedded Bug Unearthing. In *33rd USENIX Security Symposium (USENIX Security 24)*. 883–900.

[23] Juan Altmayer Pizzorno and Emery D. Berger. 2024. CoverUp: Coverage-Guided LLM-Based Test Generation. arXiv:2403.16218 [cs.SE] https://arxiv.org/abs/2403.16218

[24] Michael Pradel. 2025. Testora: Using Natural Language Intent to Detect Behavioral Regressions. arXiv:2503.18597 [cs.SE] https://arxiv.org/abs/2503.18597

[25] Binhang Qi, Yun Lin, Xinyi Weng, Yuhuan Huang, Chenyan Liu, Hailong Sun, and Jin Song Dong. 2025. Intention-Driven Generation of Project-Specific Test Cases. arXiv:2507.20619 [cs.SE] https://arxiv.org/abs/2507.20619

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

[27] Gaetano Sapia and Marcel Böhme. 2026. Scaling Security Testing by Addressing the Reachability Gap. In *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering (ICSE'26)*. 12 pages.

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

[29] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: a fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Boston, MA) *(USENIX ATC'12)*. USENIX Association, USA, 28.

[30] Prashast Srivastava and Mathias Payer. 2021. Gramatron: effective grammar-aware fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 244–256. doi:10.1145/3460319.3464814

[31] Haoxin Tu, Seongmin Lee, Yuxian Li, Peng Chen, Lingxiao Jiang, and Marcel Böhme. 2026. Cottontail: Large Language Model-Driven Concolic Execution for Highly Structured Test Input Generation. In *Proceedings of the 47th IEEE Symposium on Security and Privacy (SP'26)*. 18 pages.

[32] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2021. Unit Test Case Generation with Transformers and Focal Context. arXiv:2009.05617 [cs.SE] https://arxiv.org/abs/2009.05617

[33] Guancheng Wang, Qinghua Xu, Lionel C. Briand, and Kui Liu. 2025. Mutation-Guided Unit Test Generation with a Large Language Model. arXiv:2506.02954 [cs.SE] https://arxiv.org/abs/2506.02954

[34] Zejun Wang, Ge Li, and Zhi Jin. 2024. HITS: High-coverage LLM-based Unit Test Generation via Method Slicing. https://arxiv.org/pdf/2408.11324v1

[35] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. 2020. On learning meaningful assert statements for unit test cases. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. ACM. doi:10.1145/3377811.3380429

[36] Wikipedia contributors. 2025. Language game — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Language_game [Online; accessed 12-September-2025].

[37] Chun Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2023. Fuzz4ALL: Universal Fuzzing with Large Language Models. *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)* (2023), 1547–1559. https://api.semanticscholar.org/CorpusID:260735598

[38] Yi Xiang, Xuhong Zhang, Peiyu Liu, Shouling Ji, Hong Liang, Jiacheng Xu, and Wenhai Wang. 2024. Critical Code Guided Directed Greybox Fuzzing for Commits. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 2459–2474. https://www.usenix.org/conference/usenixsecurity24/presentation/xiang-yi

[39] Chen Yang, Junjie Chen, Bin Lin, Ziqi Wang, and Jianyi Zhou. 2025. Advancing Code Coverage: Incorporating Program Analysis with Large Language Models. *ACM Trans. Softw. Eng. Methodol.* (July 2025). doi:10.1145/3748505 Just Accepted.

[40] Han Zheng, Jiayuan Zhang, Yuhang Huang, Zezhong Ren, He Wang, Chunjie Cao, Yuqing Zhang, Flavio Toffalini, and Mathias Payer. 2023. FISHFUZZ: Catch Deeper Bugs by Throwing Larger Nets. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 1343–1360. https://www.usenix.org/conference/usenixsecurity23/presentation/zheng

[41] Xiaogang Zhu and Marcel Böhme. 2021. Regression Greybox Fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 2169–2182. doi:10.1145/3460120.3484596