# Decomposing LLM-Based Testing with Agent Skills: A Case Study on Numerical Inconsistencies

Yutong Wang
University of California, Davis
Davis, USA
ytwwang@ucdavis.edu

Cindy Rubio-González
University of California, Davis
Davis, USA
crubio@ucdavis.edu

## Abstract

LLM-based testing can expose subtle reliability issues in numerical software, but monolithic prompts make it hard to inspect which guidance drives effectiveness. We study whether Agent Skills can make such workflows more explainable by factoring procedural knowledge into composable testing components. Using numerical inconsistency detection as a case study, we decompose an LLM-based differential-testing pipeline into Skills for program generation and feedback-guided mutation, then evaluate four Skill configurations. Our results show that the feedback-guided mutation Skill setting yields the largest improvement on inconsistency rate (+12 percentage points), while enriching Skills with additional procedural knowledge slightly decreases performance (−3 percentage points). An exploratory evaluation with agent-native execution reveals scalability challenges for high-iteration Skill workflows.

## CCS Concepts

• **Software and its engineering** → **Software testing and debugging**; *Compilers*; • **Computing methodologies** → *Artificial intelligence.*

## Keywords

agent skills, explainable LLM-based testing, differential testing, numerical inconsistency detection, reliable numerical software

**ACM Reference Format:**
Yutong Wang and Cindy Rubio-González. 2026. Decomposing LLM-Based Testing with Agent Skills: A Case Study on Numerical Inconsistencies. In *Proceedings of the 2nd Workshop on Explainable and Reliable Software Systems (EXPRESS '26), October 4–9, 2026, Oakland, CA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3842650.3843176

## 1 Introduction

Reliable numerical software depends on detecting subtle compiler- and optimization-induced inconsistencies. In this domain, the same floating-point program may produce different results when compiled with different compilers or optimization levels [2, 3, 5, 10]. These inconsistencies are difficult to expose because the candidate space is large, the triggering patterns are subtle, and failures typically appear as output differences rather than crashes. LLM-based program generation is promising for this problem, but it raises an explainability question: which parts of a testing workflow make the generated programs effective?

Recent work has advanced this task through program generation and differential testing. Varity [3] uses grammar-guided random generation to synthesize floating-point programs. LLM4FP [12] instead leverages the numerical patterns LLMs have learned from real-world code to generate richer programs, detecting nearly 2.5× more numerical compiler inconsistencies than Varity. However, LLM4FP packages task specification, structural guidance, and mutation strategies into a monolithic prompt, making the LLM-driven testing process hard to inspect and ablate.

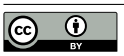
Agent Skills [1] provide a natural way to factor workflows into reusable units of procedural knowledge. A Skill is a self-contained folder consisting of task-specific instructions, optional scripts, and supporting resources. Skills can be composed into multi-step workflows and support progressive disclosure, where agents load detailed content on demand. SkillsBench [4] shows that curated Skills improve task pass rates on self-contained tasks, but how Skills behave in iterative generation settings, where feedback loops and strategy selection play a central role, remains less understood.

In this paper, we use numerical inconsistency detection as a case study to study Skill design for explainable LLM-based testing. We decompose LLM4FP into two composable Skills, one for generation and one for mutation, and vary their composition to isolate the effects of structure, feedback, and added procedural knowledge. We evaluate four configurations and report their inconsistency rates: no Skill (14.23%), one Skill (generation only, 16.47%), two Skills with feedback (28.48%), and two enriched Skills with additional procedural knowledge (25.09%). We also include an exploratory agent-native execution study via Codex CLI [8].

Our case study yields three takeaways for designing explainable LLM-based testing workflows with Agent Skills:

- **The feedback-guided mutation Skill setting has the largest effect.** Generation alone yields a modest gain (+2 percentage points), while composing it with mutation yields the largest improvement (+12 percentage points).
- **Richer Skill content is not necessarily helpful.** Enriching both Skills with floating-point procedural knowledge slightly decreases performance (−3 percentage points), suggesting detail alone does not automatically help.
- **Agent-native execution is feasible but alters costs.** A Codex CLI run generated 50 programs from a cold start with a 24.78% inconsistency rate, suggesting small-scale feasibility with shifted runtime costs.

The remainder of this paper presents our Skill design, experimental evaluation, discussion, and related work.



This work is licensed under a Creative Commons Attribution 4.0 International License.
*EXPRESS '26, Oakland, CA, USA*
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2968-3/2026/10
https://doi.org/10.1145/3842650.3843176

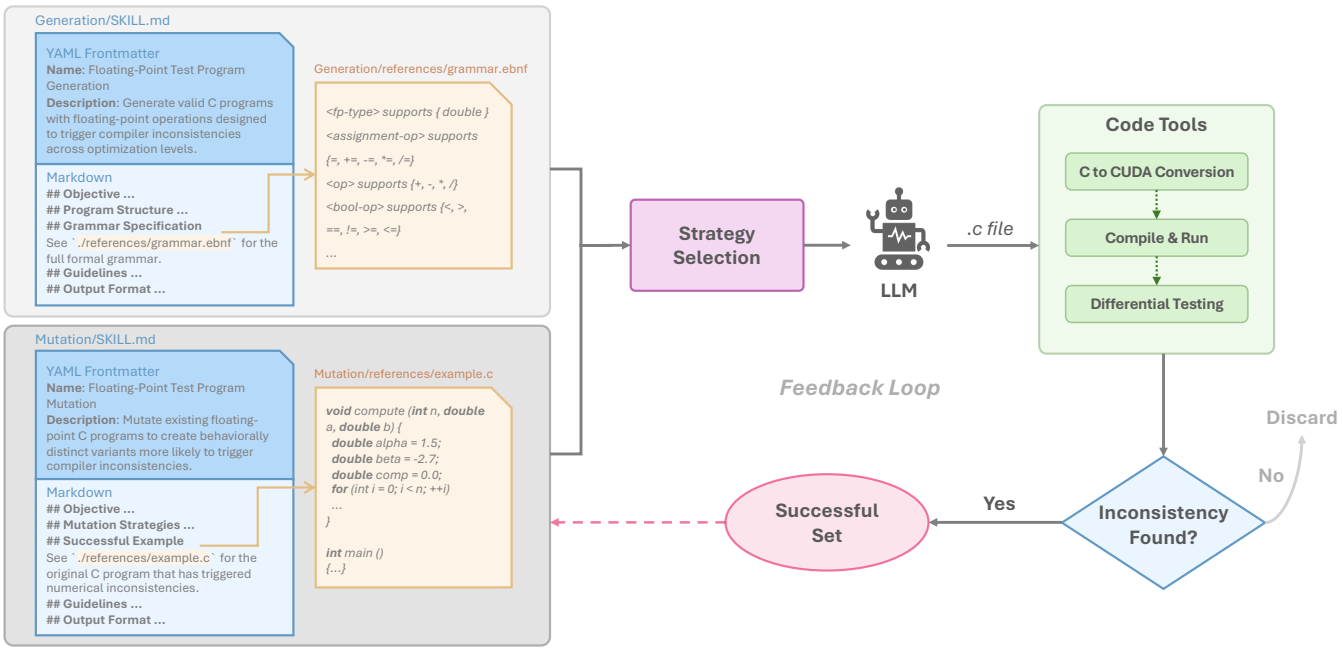



**Figure 1: Overview of the Skill-based pipeline for numerical inconsistency detection.**

## 2 Technical Approach

We study Agent Skill design in an iterative program-generation workflow for numerical inconsistency detection. Our goal is not to introduce a new differential-testing backend, but to isolate how different Skill configurations affect the quality of generated test programs. LLM4FP [12] already combines initial generation and feedback-based mutation in one workflow; we factor these tasks into separate Skills. Figure 1 shows the overall pipeline.

The pipeline operates in a loop. At each iteration, the orchestrator selects one of two strategies: generate a new program from scratch using the generation Skill, or mutate a previously successful program using the mutation Skill. The selected Skill is loaded to construct the LLM prompt, while compilation and inconsistency detection are handled by deterministic tools. If a numerical inconsistency is detected, the program is added to a successful set that can be reused in later mutation steps. This setup lets us vary Skill structure, feedback, and content while keeping the downstream testing pipeline fixed.

### 2.1 Skill Design

We represent procedural knowledge for this workflow as modular Skills, each with a `SKILL.md` file containing structured instructions, optional reference files loaded on demand, and frontmatter metadata for agent discovery. Skills can be viewed as structured prompt modules; their value here is making workflow components inspectable, reusable, and independently ablatable. This separation enables each component to be independently swapped, composed, and evaluated. We design two LLM-facing Skills:

*Generation Skill.* The objective is to generate a random, valid floating-point C program designed to expose numerical inconsistencies across compilers and optimization levels. The Skill specifies: (1) the required program structure, including a *compute* function with scalar *double*/*int* parameters storing the result in a variable named *comp*, a *dump* function for hex output, and a *main* function; (2) a grammar specification [3, 12] loaded from a reference file, derived from real-world HPC applications, that constrains the structure of the *compute* function; (3) guidelines such as using only *double* precision, avoiding undefined behavior, and initializing all variables; and (4) the output format, requiring plain source code only with no explanation or formatting.

*Mutation Skill.* The objective is to mutate an existing program that has triggered inconsistencies to create a new, behaviorally distinct variant. The Skill specifies: (1) six alternative mutation strategies, including expression transformation, control flow injection, constant perturbation, intermediate computation, operator substitution, and math function injection, from which the LLM selects one or combines a small subset based on the seed program and inconsistency report; (2) a successful program and its inconsistency report as input, encouraging the LLM to explore similar operation patterns and trigger larger inconsistencies; (3) guidelines requiring the mutated program to alter its logic and behavior (beyond simple variable renaming) while retaining its original overall structure; and (4) the output format, requiring plain source code only.

### 2.2 Code Tools

Compilation, execution, and inconsistency detection are handled by deterministic Python scripts, not by the LLM. These scripts convert C programs to CUDA for nvcc compilation, compile and run

**Table 1: Experimental configurations. Each row adds components relative to the previous one.**

| Configuration | Skills | Grammar | Mutation | Feedback |
|---|---|---|---|---|
| No Skill | 0 | × | × | × |
| Generation Only | 1 | ✓ | × | × |
| LLM4FP (2 Skills) | 2 | ✓ | ✓ | ✓ |
| + Enriched | 2 | ✓ | ✓ | ✓ |

each program across multiple compilers and optimization levels, and compare hex-encoded outputs via differential testing to detect inconsistencies. In the scripted pipeline, these tools are invoked directly by the orchestrator. In the agent-native experiment (Section 4), they are packaged as a Skill so the agent can discover and invoke them autonomously.

## 2.3 Experimental Configurations

We evaluate four configurations that progressively expose different aspects of Skill design, summarized in Table 1:

*No Skill.* The LLM receives only a basic task specification to generate a floating-point C program with a *compute* function, omitting any explicit Skill structure, mutation strategy, or feedback loop.

*Generation Only (1 Skill).* Loaded with grammar specifications and coding guidelines, the generation Skill creates each program independently from scratch, without mutation or feedback.

*LLM4FP (2 Skills).* Re-implements the LLM4FP workflow using Skills; the original LLM4FP does not use Skills. Both the Generation and Mutation Skills are loaded. The pipeline maintains a set of successful programs and probabilistically chooses between generating from scratch and mutating a randomly selected successful program. This condition adds feedback-guided Skill composition on top of structured generation.

*Enriched Skills (2 Skills + Detailed Knowledge).* This configuration keeps the same two-Skill composition and feedback loop, but enriches both Skills with additional procedural knowledge about floating-point inconsistency detection. The Generation Skill is augmented with procedural knowledge about constructs that are more likely to trigger inconsistencies, while the Mutation Skill is augmented with procedural knowledge for diagnosing inconsistency causes and selecting effective mutations.

# 3 Experimental Evaluation

This evaluation measures how Skill structure, feedback-guided mutation, and added procedural knowledge affect inconsistency detection effectiveness and runtime.

## 3.1 Experimental Setup

All four LLM-based methods use GPT-4.1 [7] and generate 1,000 floating-point programs in C. Each program is compiled with gcc 9.4, clang 12.0, and nvcc 12.3 at six optimization levels (O0_nofma, O0, O1, O2, O3, O3_fastmath), producing 18 compiler-pair comparisons per program and 18,000 comparisons per method. We report

**Table 2: Inconsistency rates, counts, and time costs across configurations with different Skill structures and content.**

| Configuration | Skills | Incon. Rate | # Incons. | Time Cost |
|---|---|---|---|---|
| Varity | — | 11.93% | 2,147 | **00:30:42** |
| No Skill | 0 | 14.23% | 2,562 | 04:16:37 |
| Generation Only | 1 | 16.47% | 2,964 | 05:56:13 |
| LLM4FP (2 Skills) | 2 | **28.48%** | **5,126** | 04:23:04 |
| + Enriched | 2 | 25.09% | 4,516 | 05:11:04 |

inconsistency rate, number of inconsistent comparisons, and wall-clock time, and use the random program generator Varity [3] as baseline. All experiments run on Ubuntu 20.04 with x86_64.

## 3.2 Results

Table 2 shows the results for each Skill configuration. We make three key observations:

*The Feedback-Guided Mutation Skill Setting Has the Largest Effect.* Each additional Skill improves the inconsistency rate: 14.23% (no Skill) → 16.47% (1 Skill) → 28.48% (2 Skills). The largest gain comes from composing the generation Skill with a mutation Skill that creates a feedback loop over successful programs (+12.01 percentage points), while adding the generation Skill alone provides a more modest improvement (+2.24 percentage points).

*Richer Skill Content Does Not Improve Performance in This Setting.* Holding the two-Skill composition and feedback loop fixed, enriching both Skills with additional procedural knowledge reduces the inconsistency rate from 28.48% to 25.09%. This suggests that increasing Skill content does not necessarily improve an iterative generation workflow, even when the added content is task-relevant procedural knowledge. One possible explanation is that the added knowledge narrows the model's exploration, causing it to overuse a smaller set of familiar constructs rather than explore a broader program space. Another possibility is that the specific knowledge we added was not well-calibrated enough for this task. The enriched configuration also takes longer (5:11:04 vs. 4:23:04) and yields fewer inconsistencies overall (4,516 vs. 5,126).

*LLM-Based Generation Trades Runtime for Richer Exploration.* Random program generation (Varity) remains an efficient and strong baseline, reaching an 11.93% inconsistency rate in about 30 minutes. The LLM-based configurations explore richer numerical code patterns and improve inconsistency rates, but require 4–6 hours, so the benefit comes with a higher runtime cost.

*Agent-Native Execution.* We conducted an isolated Codex CLI [8] experiment to examine whether an agent can execute the Skill-based workflow from a cold start. The run used GPT-5.5 [9] with medium reasoning effort. The agent was given only the Skill instructions and deterministic testing tools, with no access to prior generated programs, cached results, or successful examples. It generated 50 C floating-point programs and tested them with gcc, clang, and nvcc across six optimization levels. All 50 programs compiled successfully. Using the same pairwise-comparison definition as the main evaluation, the run produced 223 inconsistent comparisons out of 900 total comparisons, yielding a 24.78% inconsistency rate.

This result suggests that agent-native execution is a viable way to instantiate Skill workflows at small scale. In this setting, the agent produced high-quality test programs from a cold start, with an inconsistency rate close to the scripted enriched-Skill configuration. The run took about 19 minutes for 50 programs, which extrapolates to roughly 6.4 hours for 1,000 programs before accounting for additional long-run overhead. This is just over two hours slower than the scripted Skill-composition run, which completed 1,000 programs in 4:23:04, suggesting that agent-native execution trades off automation flexibility against higher per-program orchestration cost. Because this runtime includes model choice, planning, tool invocation, and orchestration overhead, it should be read as exploratory rather than as a controlled cost comparison.

## 4 Discussion

*Skill Composition vs. Content.* Our results suggest that how Skills are connected can matter as much as their content. The mutation Skill improves performance by reusing programs that already exposed inconsistencies and asking the model to create variants, giving the workflow a simple feedback loop. By contrast, adding more floating-point knowledge did not improve performance in our experiments. Whether this is because the specific knowledge we added was not well-suited for this task remains an open question. SkillsBench [4] finds that curated Skills improve performance on self-contained tasks; our setting suggests that future work should study what types of Skill knowledge are most effective for iterative generation workflows, and how Skills interact over time.

*Explainability Through Workflow Decomposition.* In this work, explainability does not mean explaining individual LLM token choices. Instead, we treat explainability as the ability to inspect, separate, and evaluate the procedural components of an LLM-based testing workflow. The Skill decomposition exposes which guidance is used for initial generation, which guidance is used for mutation, where feedback enters the loop, and which parts are handled by deterministic testing tools rather than the LLM. This structure makes the workflow easier to audit than a monolithic prompt and enables the ablation in Table 2: we can attribute the largest improvement to feedback-guided mutation rather than to added prompt detail alone. This workflow-level view is especially useful for reliable software testing, where effectiveness, cost, and reproducibility all matter.

*Threats to Validity.* Our main evaluation uses one LLM (GPT-4.1) and one 1,000-program run per method, so results may vary across models, prompts, and repeated runs. Small gaps such as *Varity* vs. *No Skill* vs. *Generation Only* should therefore be interpreted cautiously; the main qualitative observation is the larger jump from feedback-guided mutation. The enriched knowledge was manually curated and may not be the best procedural knowledge for this task. The agent-native experiment uses Codex CLI with GPT-5.5 and medium reasoning effort, so it is exploratory and not directly comparable to the scripted GPT-4.1 runs. Our findings may generalize mainly to iterative workflows with reusable successful artifacts and feedback, not necessarily all LLM-based testing tasks.

*Future Directions.* Future work should study how to calibrate procedural knowledge inside Skills: what knowledge should be included, how specific it should be, and when added detail narrows exploration instead of improving results. Another direction is to decompose the workflow into finer-grained Skills, such as strategy selection, inconsistency diagnosis, mutation planning, and result summarization, and then study how these Skills interact. Such designs could make explainability a first-class evaluation target: beyond measuring inconsistency rates, systems could expose traces showing which Skill was selected, which prior failures informed mutation, and which deterministic checks accepted or rejected each program. Future evaluations should also repeat configurations across models and random seeds while reporting token cost, wall-clock time, compilation failures, and orchestration overhead.

## 5 Related Work

*Numerical Inconsistency Detection.* Varity [3] uses random generation to synthesize floating-point programs for differential testing. LLM4FP [12] extends this with LLM-based generation and feedback-based mutation. FPDiff [11] and FLiT [10] perform differential testing on numerical libraries and across platforms, respectively. pLiner [2] and Ciel [5] localize inconsistencies at the line and expression level. Miao et al. [6] generate input ranges that expose compiler-induced numerical inconsistencies. Our work is complementary: we study how to package and compose LLM guidance for iterative program generation.

*LLM-Based Program Generation for Testing.* Recent work has applied LLMs to compiler testing more broadly. WhiteFox [15] uses LLMs for white-box compiler fuzzing by generating programs that target specific optimization passes. Fuzz4ALL [13] applies LLMs to universal fuzzing across multiple language frontends. These systems use LLMs to generate tests directly; our work studies how to factor generation strategies into reusable Skills.

*Agent Skills.* Agent Skills [1] are structured packages of procedural knowledge for LLM agents. SkillsBench [4] provides the first systematic evaluation of Skills across diverse tasks, showing that curated Skills substantially improve performance while self-generated Skills show minimal benefit. Xu and Yan [14] survey the landscape of Agent Skills, covering architectural foundations, Skill acquisition, deployment, and security. Our work extends this line to iterative generation workflows, where composed Skills and feedback can shape later generation steps.

## 6 Conclusion

We studied Agent Skill design for explainable LLM-based testing using numerical inconsistency detection as a case study. Our results show that the feedback-guided mutation Skill setting yields the largest improvement: adding a mutation Skill creates a feedback loop that builds upon successful programs, while adding more procedural knowledge to Skills does not automatically help. An exploratory Codex CLI experiment further suggests that agent-native Skill workflows can produce useful programs from a cold start, but their runtime and orchestration costs should be reported alongside effectiveness. Overall, these findings show that high-iteration agent workflows for reliable software testing require evaluating both how Skills are composed and whether their content is calibrated to the iterative task.

## Acknowledgments

This work was supported in part by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research under award DE-SC0025922, and the National Science Foundation under awards CNS-2346396 and CCF-2446223.

## References

[1] Anthropic. 2025. Equipping Agents for the Real World with Agent Skills. https://www.anthropic.com/engineering/equipping-agents-for-the-real-world-with-agent-skills.

[2] Hui Guo, Ignacio Laguna, and Cindy Rubio-González. 2020. pLiner: isolating lines of floating-point code for compiler-induced variability. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, Christine Cuicchi, Irene Qualters, and William T. Kramer (Eds.). IEEE/ACM, 49. doi:10.1109/SC41405.2020.00053

[3] Ignacio Laguna. 2020. Varity: Quantifying Floating-Point Variations in HPC Systems Through Randomized Testing. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), New Orleans, LA, USA, May 18-22, 2020*. IEEE, 622–633. doi:10.1109/IPDPS47924.2020.00070

[4] Xiangyi Li, Wenbo Chen, Yimin Liu, Shenghan Zheng, Xiaokun Chen, Yifeng He, Yubo Li, Bingran You, Haotian Shen, Jiankai Sun, Shuyi Wang, Qunhong Zeng, Di Wang, Xuandong Zhao, Yuanli Wang, Roey Ben Chaim, Zonglin Di, Yipeng Gao, Junwei He, Yizhuo He, Liqiang Jing, Luyang Kong, Xin Lan, Jiachen Li, Songlin Li, Yijiang Li, Yueqian Lin, Xinyi Liu, Xuanqing Liu, Haoran Lyu, Ze Ma, Bowei Wang, Runhui Wang, Tianfu Wang, Wengao Ye, Yue Zhang, Hanwen Xing, Yiqi Xue, Steven Dillmann, and Han-Chung Lee. 2026. SkillsBench: Benchmarking How Well Agent Skills Work Across Diverse Tasks. *CoRR* abs/2602.12670 (2026). arXiv:2602.12670 doi:10.48550/ARXIV.2602.12670

[5] Dolores Miao, Ignacio Laguna, and Cindy Rubio-González. 2023. Expression Isolation of Compiler-Induced Numerical Inconsistencies in Heterogeneous Code. In *High Performance Computing - 38th International Conference, ISC High Performance 2023, Hamburg, Germany, May 21-25, 2023, Proceedings (Lecture Notes in Computer Science, Vol. 13948)*, Abhinav Bhatele, Jeff R. Hammond, Marc Baboulin, and Carola Kruse (Eds.). Springer, 381–401. doi:10.1007/978-3-031-32041-5_20

[6] Dolores Miao, Ignacio Laguna, and Cindy Rubio-González. 2024. Input Range Generation for Compiler-Induced Numerical Inconsistencies. In *Proceedings of the 38th ACM International Conference on Supercomputing, ICS 2024, Kyoto, Japan, June 4-7, 2024*, Kenji Kise, Valentina Salapura, Murali Annavaram, and Ana Lucia Varbanescu (Eds.). ACM, 201–212. doi:10.1145/3650200.3656618

[7] OpenAI. 2023. GPT-4 Technical Report. *CoRR* abs/2303.08774 (2023). arXiv:2303.08774 doi:10.48550/ARXIV.2303.08774

[8] OpenAI. 2026. Codex CLI. https://github.com/openai/codex.

[9] OpenAI. 2026. Introducing GPT-5.5. https://openai.com/index/introducing-gpt-5-5/. Accessed: 2026-04-30.

[10] Geoffrey Sawaya, Michael Bentley, Ian Briggs, Ganesh Gopalakrishnan, and Dong H. Ahn. 2017. FLiT: Cross-platform floating-point result-consistency tester and workload. In *2017 IEEE International Symposium on Workload Characterization, IISWC 2017, Seattle, WA, USA, October 1-3, 2017*. IEEE Computer Society, 229–238. doi:10.1109/IISWC.2017.8167780

[11] Jackson Vanover, Xuan Deng, and Cindy Rubio-González. 2020. Discovering discrepancies in numerical libraries. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, Sarfraz Khurshid and Corina S. Pasareanu (Eds.). ACM, 488–501. doi:10.1145/3395363.3397380

[12] Yutong Wang and Cindy Rubio-González. 2025. LLM4FP: LLM-Based Program Generation for Triggering Floating-Point Inconsistencies Across Compilers. In *Proceedings of the SC '25 Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC Workshops 2025, St Louis, MO, USA, November 16-21, 2025*. ACM, 225–234. doi:10.1145/3731599.3767362

[13] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4All: Universal Fuzzing with Large Language Models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 126:1–126:13. doi:10.1145/3597503.3639121

[14] Renjun Xu and Yang Yan. 2026. Agent Skills for Large Language Models: Architecture, Acquisition, Security, and the Path Forward. *CoRR* abs/2602.12430 (2026). arXiv:2602.12430 doi:10.48550/ARXIV.2602.12430

[15] Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. 2024. WhiteFox: White-Box Compiler Fuzzing Empowered by Large Language Models. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024), 709–735. doi:10.1145/3689736

Received 2026-06-28; accepted 2026-07-31