Towards Safe Agentic AI Performance Engineering

Dan Williams
Virginia Tech, IBM
Blacksburg, VA, USA
djwillia@vt.edu

Milo Craun
Virginia Tech
Blacksburg, VA, USA
miloc@vt.edu

Michael V. Le
IBM
Yorktown Heights, NY, USA
mvle@us.ibm.com

Julian Stephen
IBM
Yorktown Heights, NY, USA
julian.stephen@ibm.com

Salman Ahmed
IBM
Yorktown Heights, NY, USA
sahmed@ibm.com

Hani Jamjoom
IBM
Yorktown Heights, NY, USA
jamjoom@us.ibm.com

Abstract

The emergence of agentic AI—reasoning AI agents that can connect to tools and take actions—offers an enormous potential in performing tasks that currently require highly skilled humans to perform. In this position paper, we discuss AI agents in one such role: *performance engineer*. A performance engineer is typically highly trained and highly trusted to run performance diagnostic tools—which more often than not require root or administrator privileges—on production machines to diagnose performance issues. Critically, performance engineers are trusted not to cause harm to the production systems they are investigating, including crashing or hanging the systems, extracting sensitive information from them, or negatively affecting their performance. In this paper, we argue that current AI agents have the training, but lack the trust to be performance engineers. We outline four components: prevention, detection/auditing, aborting/rollback, and retry/refocus and highlight gaps where the approaches taken for human-based performance engineers fall short.

CCS Concepts: • **Security and privacy** → *Software security engineering*.

Keywords: Agentic AI, Agents, Performance engineering, Trust, AI systems

ACM Reference Format:
Dan Williams, Milo Craun, Michael V. Le, Julian Stephen, Salman Ahmed, and Hani Jamjoom. 2025. Towards Safe Agentic AI Performance Engineering. In *Practical Adoption Challenges of ML for Systems (PACMI '25), October 13–16, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3766882.3767179

PACMI '25, October 13-16, 2025, Seoul, Republic of Korea

ACM ISBN 979-8-4007-2205-9/25/10
https://doi.org/10.1145/3766882.3767179

1 Introduction

Agentic AI, or groups of AI agents that can connect to our systems and carry out actions, promises to change the way we manage our systems. One role that has generally required highly skilled humans is that of a performance engineer. A performance engineer must diagnose performance issues in production systems. Performance issues may be subtle, unique, novel, and infrequent; they may involve any component in the complex system stack or interactions between components, and the root cause may be obscured by various symptoms. In addition to general systems knowledge, the performance engineer must make hypotheses, utilize or create tools to gather data to test those hypotheses, and analyze results until a conclusion is made. Further complicating matters is the fact that the experiments performed must not negatively impact likely production workloads.

Due to the nature of their tasks, especially the ability to observe any component in the system (or their interactions), performance engineers require high privilege in the system and are trusted not to abuse this privilege. This is in contrast to systems like ChatDBG [26] or CoverUp [14] that connect LLMs to existing debugging tools that do not require privileged access to production systems. We characterize our trust in performance engineers into two aspects: *safety* and *liveness*. Performance engineers are trusted to maintain safety: they must not crash systems, degrade performance, or steal any of the potentially sensitive information they have access to in their role. Obviously we also expect liveness: any performance engineer worth their title should be able to make progress and diagnose issues without hallucinating or getting stuck in a loop.

In this paper, we examine how AI agents fare as performance engineers and outline the systems that must be in place for them to achieve the properties of safety and liveness. We first describe our experience using an LLM-based chatbot to debug a performance issue with a human-mediated connection to the system, concluding that they are already capable of performing the basic steps of a performance engineer. We then identify the systems components required to take the human out of the loop.

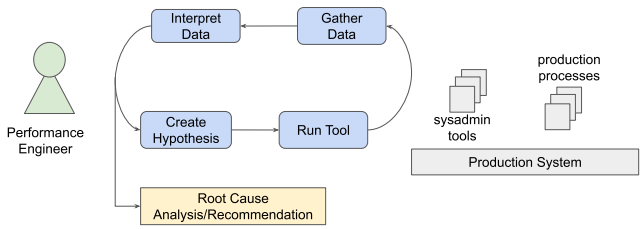

Figure 1. Performance engineering workflow overview

We organize the required system components into four categories: prevention, detection/auditing, aborting/rollback, and retry/refocus. We discuss prevention, both via prompt-based techniques and sandboxing techniques including verification for agent-written probe programs. Then, we describe the importance of detection, monitoring, and auditing on the target systems and ensuring the information feeds back into the agent. We outline a requirement for aborting commands and rolling back their effects. Finally, we describe techniques to keep the AI performance engineer on track.

2 An AI Performance Engineer is Born

A performance engineer is generally tasked with debugging performance issues on production systems, as shown in Figure 1. Given a production system running production workloads, this task involves three iterative steps:

1. **Create Hypothesis:** based on data gathered about the system so far, the performance engineer hypothesizes a cause of the problem.
2. **Run Tools to Gather Data:** in order to prove or disprove the hypothesis, and to aid in creating future hypotheses, the performance engineer runs system administration tools to gather data on the system. These tools generally require privilege as they may need to interpose on sensitive parts of the system—both in terms of performance sensitivity and data sensitivity—such as the OS kernel.
3. **Interpret Data:** based on the results of the tools, the performance engineer may rule out a hypothesized root cause or otherwise use the new data to create or refine the hypothesis, in either case returning to step 1. Alternatively, the performance engineer may ultimately confirm a root cause, providing a recommendation to developers (with accompanying analysis to justify the recommendation).

In the rest of this section, we describe a simple performance engineering task, and how ChatGPT can already debug the root cause of the issue. For this section, the application we are debugging is a logging application[1] that calls `fsync()` after every write—regardless of its size—with no buffering. This results in a large amount of I/O waiting and poor latency.

[1]Example inspired by Brendan Gregg's BPF perf workshop lab001, 2019. [23]

2.1 Experimental Setup

We set up the experiment as follows. We run the poorly performing logging application on a local laptop with 8 Intel i7 2.8GHz CPUs, 64 GB RAM, and a 500GB NVMe Samsung SSD. The laptop has a full suite of performance engineering tools installed, including `iotop`, `bpftrace` and others. Most performance engineering tools require root access, provided via `sudo` and some—particularly `bpftrace`—accept custom user programs to create one-off custom tools, in a similar way as shell scripts.

For the AI, we utilize the free tier of the default ChatGPT based on GPT-4, specifically the GPT-4o variant (as of June 2024). We utilized a manual connector between ChatGPT and the system; in other words, we copy/pasted command line instructions from the chat window to the terminal and vice-versa for output. In future work, we will automate this process with a reasonable protocol, such as the Model Context Protocol (MCP).

For the initial prompt, we used:

> We are debugging an application that has higher-than expected latency, including latency outliers. We want to know why, and how can performance be improved? The system has all of the tools for debugging the issue installed. Tell me what to type into the command line and I will give you the output of the command.

2.2 Performance Debugging with ChatGPT

ChatGPT successfully debugged the performance issue using a series of tools, summarized in Table 1. Some steps it took were superfluous, but it did manage to iteratively refine hypotheses until determining that the application performed too many `fsync` operations, recommending batching writes, alternately configuring fsync or using alternatives with more control over writeback.

ChatGPT begun by examining whether the system was overloaded, using `uptime`. After seeing that the system was not overloaded, it next investigated whether processes were blocked on the system using `vmstat`, finding non-trivial I/O waiting. It then identified the process that was most responsible for I/O (via writes) using `iotop`, and counted how many write system calls were being performed and their latency distribution with two different `bpftrace` custom scripts. When no obvious issues were found with writes, ChatGPT took a misstep by hypothesizing reads were the issue, but rejected that hypothesis after observing no reads, again using `bpftrace`. Next, ChatGPT counted all system calls being performed by the process in question with `bpftrace`, finding lots of calls to `fsync`. After confirming latency outliers up to 2-4 seconds from fsync, ChatGPT generated recommendations for fixing the program as described above.

This example demonstrates that ChatGPT does have the capability to perform the iterative performance engineering

Hypothesis	Tool	Data Gathered	Insight	Priv?
System is overloaded	`uptime`	load average	system not overloaded	No
Something is blocking	`vmstat`	blocked processes, I/O wait	non-trivial I/O waiting	No
Something is waiting on disk I/O	`iotop`	pid responsible for I/O	pid, mostly writes	Yes
Pid is writing a lot	`bpftrace`	count write syscalls	lots of writes	Yes
Pid's writes are slow	`bpftrace`	latency histogram of writes	writes are fast	Yes
Pid's reads are slow	`bpftrace`	latency histogram of reads	no reads	Yes
Pid is doing another slow I/O syscall	`bpftrace`	all system calls	lots of fsyncs	Yes
Pid's fsyncs are slow	`bpftrace`	latency histogram of fsync	up to 2-4 seconds	Yes

Table 1. Hypotheses and system tools used by ChatGPT during performance debugging experiment

workflow. It generated hypotheses and determined the appropriate tools to run/data to gather to confirm or evolve those hypotheses. In the next section, we explore the question of whether it should be trusted to run these tools.

3 We Trust Performance Engineers

Performance engineers are generally granted administrator-level permissions on production systems in order to run the relevant tools, some of which are not known in advance (e.g., custom scripts). A competent performance engineer can diagnose issues on these systems while maintaining two properties: *safety* and *liveness*. In this section, we discuss threats to each in turn.

Safety. Exacerbated by administrator-level privileges, we are concerned with three broad threat classes from an errant performance engineer:

- **Crashing systems:** With administrative privileges, it becomes trivial to crash an application or an entire system, for instance by sending a process a signal or installing a kernel module that immediately panics the kernel on purpose or inadvertently (e.g., through a null pointer dereference). While many existing tools will be trusted not to crash systems, custom scripts or modules developed by the performance engineer introduce a higher likelihood of crashing systems.
- **Degrading performance:** The performance engineer necessarily runs tools to observe the system. Many tools can dramatically impact the performance of the target system and are frequently avoided for in-production debugging (e.g., `ptrace`-based tools including `strace`).
- **Stealing information:** Via running observability tools with administrator-level privileges, the performance engineer has access to sensitive data in the system, including access to process memory, cryptographic keys, user data, etc. Leaking this data constitutes a major compromise of a system.

Liveness. By liveness, we refer to the performance engineer's ability to make progress and solve problems. We consider the following threats to liveness:

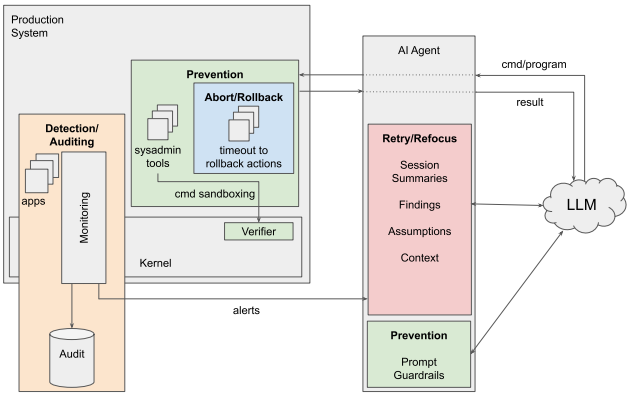

Figure 2. System overview

- **Stuck tools:** Sometimes, the tools used to observe systems can themselves not make progress, exhibiting behavior like infinite loops, even as the system continues to operate correctly. Tools may wait for an event which will never happen due to an incorrect hypothesis about the system state. As a concrete example, consider a tool that monopolizes the shell; performance engineers must know how to regain control.
- **Hallucination:** While the term "hallucination" is rarely applied to human performance engineers, the phenomenon of taking action based on incorrect assumptions, leading to incorrect diagnoses and the design of irrelevant experiments is common, especially for less experienced engineers.
- **Looping:** We refer to a specific form of hallucination as *looping*, in which a performance engineer enters a loop of actions, repeatedly trying the same experiment, or switching between a few experiments, not making progress on identifying the root cause. In dynamic systems, where the same experiment may produce different numbers even while showing the same trend.

4 Enabling AI Performance Engineers

In order for AI agents to be performance engineers we must ensure that they are able to achieve both safety and liveness.

We advocate that the technical aspects of threat mitigations (such as system sandboxes, monitoring, auditing, etc.) utilized by current human performance engineers continue to be strengthened and enhanced, and highlight areas where AI agent specific policies are needed. We present a proposed system overview in Figure 2. A core AI agent workflow is augmented using four techniques: *Prevention*, *Detection and Auditing*, *Aborting and Rollback*, and *Retry and Refocus*. While the former two techniques are largely similar for human and AI performance engineers, the latter two are more unique to AI performance engineers and subsequently lack the years of development and maturity of the former. We now describe each technique in more detail.

Prevention. The first strategy is to attempt to prevent safety or liveness violations before they happen. In current systems with human performance engineers, technical system safeguards, such as sandboxing can mitigate issues. For example, the power of "the omnipotent root" can be managed via jails [25], zones [30], or container configurations that contain curated lists of safe tools.[2] Similarly, finer-grained policies utilizing role-based access control [22] and system configurations like SELinux [27] or the use of sub-root capabilities [15] can be used to implement least-privilege sandboxes. More recently, technical sandboxing approaches utilizing bytecode verification (e.g., BPF) have become popular, in which probes installed by a benign but error prone performance engineer are verified before executing in the kernel. Note that these sandboxes are not perfect: a malicious BPF user can crash a system despite the verifier with a variety of techniques, such as stack overflow [33], interface misuse [24], and verifier exploits [1–4, 35]. Improvement of sandboxes is an active area of research in computer systems; for example, in the context of BPF, work is ongoing to improve BPF's guarantees to prevent crashes [16, 17], to prevent performance degradation [19, 31, 32], and to prevent information stealing [21].

Although many of these systems sandboxing approaches were designed with human performance engineers in mind, they are also useful for AI performance engineers. Specific to agents, E2B [6] utilizes lightweight VMs [13] as a sandbox for agents or tools, while FIDES adds an information flow control component to data input and output from the tools used by agents [18]. Furthermore, AI-specific model and prompt mitigations that involve training or fine-tuning the model [29], prompt engineering [5, 34] and adding guardrails [7, 8, 10, 20] around inputs and outputs of the model have been explored to avoid problematic outcomes. For example, the model may be prompted to avoid suggestions that use expensive monitoring tools like `strace` and to instead prefer `bpftrace` (as in the preceding example). Such mitigation strategies are imprecise, but are necessary as they capture intuition well.

[2]For examples, see https://hub.docker.com/search?q=debug

Detection and Auditing. Distributed systems monitoring [28] is a critical component of modern deployments, with a range of tools including Prometheus, Grafana, OpenTelemetry, etc. providing dashboards that convey the health of the system and applications running therein. For human performance engineers, these systems provide two services: 1) they give feedback to the performance engineer on the impact of running diagnostic tools on the system, and 2) they log the performance engineers actions, providing a disincentive to deliberately cause harm to the system via purposely crashing systems, degrading performance, or steal information, since review of the logs carries a threat of losing employment or legal action.

For AI performance engineers, monitoring and auditing is also important [12]. Access to monitoring metrics provides additional signals for the AI agent, and may provide additional context to help diagnose performance problems. A running auditing system that records performance metrics and AI agent actions is important to allow for transparency in the effectiveness of an AI performance engineer. External system operators need indications of the success (or failure) of the AI in order to decide if the agent should be deployed or if it is not functioning properly.

Aborting and Rollback. In current systems with human performance engineers, threats to liveness are closely related to engineer competence and coarsely measured human processes like annual performance reviews. Issues are generally mitigated through training, experience and best practices. A trained human performance engineer is unlikely to not know how to recover from a stuck tool and regain control. Similarly, the usage of monitoring tools for feedback allows the human to determine if performance degradation has occurred, but the human is expected to know how to undo whatever they did to cause the degradation.

However, we identify a gap: current descriptions of tools and ways to connect AI agents to them [9] do not contain instructions about what to do to if an tool gets stuck or an agent forgets what it did to cause a performance degradation. We propose that the underlying system and tools must support rollback operations, or risk safety and liveness violations. As a start, tools should implement timeouts and cancellations to support aborting and rollback.

Retry and Refocus. Similar to the above, in current systems with human performance engineers, threats to liveness related to lack of focus are closely related to engineer competence and coarsely measured human processes like annual performance reviews. Issues are generally mitigated through training, experience and best practices. Human performance engineers are expected not to forget their task!

As described, hallucinations and looping behavior are threats to liveness that do not have technical solutions in the human-based world. However, we believe that many of the strategies used by human performance engineers to be

effective can be applied. For instance, education about tool usage and the practice of note keeping and writing down assumptions can be incorporated into prompts to encourage an LLM to stay focused on the task at hand. There are efforts in this area: special prompts including few shot prompt or chain of thought and more task oriented dialogue management systems, such as Rasa [11], can act as a wrapper around the LLM to filter and/or track conversation state in a specified way to keep the LLM on track. Agents can also leverage their own memory to detect repeated patterns and gauge progress.

5 Summary

In this work we posit that while existing AI models already have the knowledge to debug performance issues in production systems, they currently lack the necessary trust to be deployed on a production system as a performance engineer. We advocate for continued work on prevention with traditional systems sandboxing, including verification of custom code as in BPF, as well as augmented prevention via prompt-based methods. We advocate for continued work on traditional monitoring, with dashboard outputs fed back into the AI agent. We advocate augmenting tool descriptions and implementations with rollback capabilities. Finally, we advocate dialogue management to keep AI agents on track. With these components, AI performance engineers can debug novel subtle performance bugs in production systems while maintaining safety and liveness.

References

[1] 2021. CVE-2021-31440. https://nvd.nist.gov/vuln/detail/CVE-2021-31440.

[2] 2021. CVE-2021-45402. https://nvd.nist.gov/vuln/detail/CVE-2021-45402.

[3] 2022. CVE-2022-23222. https://nvd.nist.gov/vuln/detail/CVE-2022-23222.

[4] 2022. CVE-2022-2785. https://nvd.nist.gov/vuln/detail/CVE-2022-2785.

[5] 2025. DSPy. https://dspy.ai/.

[6] 2025. E2B. https://e2b.dev/docs.

[7] 2025. Guardrails AI. https://www.guardrailsai.com/docs.

[8] 2025. LLM Guard. https://github.com/protectai/llm-guard.

[9] 2025. Model Context Protocol. https://modelcontextprotocol.io/overview.

[10] 2025. NVIDIA Nemo Guardrails. https://github.com/NVIDIA/NeMo-Guardrails.

[11] 2025. Rasa. https://github.com/rasahq.

[12] 2025. What is AI Monitoring? https://www.trustible.ai/post/what-is-ai-monitoring.

[13] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. Santa Clara, CA.

[14] Juan Altmayer Pizzorno and Emery D. Berger. 2025. CoverUp: Effective High Coverage Test Generation for Python. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE128 (June 2025), 23 pages. doi:10.1145/3729398

[15] Serge E. Bacarella. 2000. POSIX Capabilities. In *Proceedings of the 2000 USENIX Annual Technical Conference: FREENIX Track*. USENIX Association, San Diego, CA, USA, 29–36.

[16] Siddharth Chintamaneni. 2025. *Unsafe Nesting in BPF Programs.* Master's thesis. Virginia Polytechnic Institute and State University.

[17] Siddharth Chintamaneni, Sai Roop Somaraju, and Dan Williams. 2024. Unsafe kernel extension composition via BPF program nesting. In *Proceedings of the ACM SIGCOMM 2024 Workshop on EBPF and Kernel Extensions* (Sydney, NSW, Australia) *(eBPF '24)*. Association for Computing Machinery, New York, NY, USA, 65–67. doi:10.1145/3672197.3673440

[18] Manuel Costa, Boris Köpf, Aashish Kolluri, Andrew Paverd, Mark Russinovich, Ahmed Salem, Shruti Tople, Lukas Wutschitz, and Santiago Zanella-Béguelin. 2025. Securing AI Agents with Information-Flow Control. arXiv:2505.23643 [cs.CR] https://arxiv.org/abs/2505.23643

[19] Milo Craun, Khizar Hussain, Uddhav Gautam, Zhengjie Ji, Tanuj Rao, and Dan Williams. 2024. Eliminating eBPF Tracing Overhead on Untraced Processes. In *Proceedings of the ACM SIGCOMM 2024 Workshop on EBPF and Kernel Extensions* (Sydney, NSW, Australia) *(eBPF '24)*. Association for Computing Machinery, New York, NY, USA, 16–22. doi:10.1145/3672197.3673431

[20] Edoardo Debenedetti, Ilia Shumailov, Tianqi Fan, Jamie Hayes, Nicholas Carlini, Daniel Fabian, Christoph Kern, Chongyang Shi, Andreas Terzis, and Florian Tramèr. 2025. Defeating Prompt Injections by Design. arXiv:2503.18813 [cs.CR] https://arxiv.org/abs/2503.18813

[21] Chinecherem Stephanie Dimobi. 2025. *Preventing Unintended Data Access: Information Flow Control in eBPF.* Master's thesis. Virginia Polytechnic Institute and State University.

[22] David F. Ferraiolo and D. Richard Kuhn. 1992. Role-Based Access Controls. In *Proceedings of the 15th National Computer Security Conference (NCSC)*. National Institute of Standards and Technology (NIST), 554–563.

[23] Brendan Gregg. 2019. BPF Performance Tools Workshop. https://github.com/brendangregg/bpf-perf-workshop.

[24] Jinghao Jia, Raj Sahu, Adam Oswald, Dan Williams, Michael V. Le, and Tianyin Xu. 2023. Kernel extension verification is untenable. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems* (Providence, RI, USA) *(HotOS '23)*. Association for Computing Machinery, New York, NY, USA, 150–157. doi:10.1145/3593856.3595892

[25] Poul-Henning Kamp and Robert N. M. Watson. 2000. Jails: Confining the Omnipotent Root. In *Proceedings of the 2nd International System Administration and Networking (SANE 2000) Conference*. NLUUG, Maastricht, The Netherlands.

[26] Kyla H. Levin, Nicolas van Kempen, Emery D. Berger, and Stephen N. Freund. 2025. ChatDBG: Augmenting Debugging with Large Language Models. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE085 (June 2025), 22 pages. doi:10.1145/3729355

[27] Peter A. Loscocco and Stephen D. Smalley. 2001. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*. USENIX Association, Boston, MA, USA, 29–42.

[28] Matthew L Massie, Brent N Chun, and David E Culler. 2004. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Comput.* 30, 7 (2004), 817–840. doi:10.1016/j.parco.2004.04.001

[29] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. arXiv:2203.02155 [cs.CL] https://arxiv.org/abs/2203.02155

[30] Daniel Price and Andrew Tucker. 2004. Solaris Zones: Operating System Support for Consolidating Commercial Workloads. In *Proceedings of the 18th Large Installation System Administration Conference (LISA 2004)*. USENIX Association, Atlanta, GA, USA, 241–254.

[31] Raj Sahu and Dan Williams. 2023. Enabling BPF Runtime policies for better BPF management. In *Proceedings of the 1st Workshop on EBPF and Kernel Extensions* (New York, NY, USA) *(eBPF '23)*. Association for Computing Machinery, New York, NY, USA, 49–55. doi:10.1145/3609021.3609297

[32] Raj Sahu and Dan Williams. 2023. When BPF programs need to die: exploring the design space for early BPF termination. In *Linux Plumbers Conference (LPC'23)*. https://lpc.events/event/17/contributions/1610/. (Nov. 2023).

[33] Sai Roop Somaraju, Siddharth Chintamaneni, and Dan Williams. [n. d.]. Overflowing the kernel stack with BPF. In *Linux Plumbers Conference (LPC'23)*. https://lpc.events/event/17/contributions/1595/. (Nov. 2023).

[34] Mandana Vaziri, Louis Mandel, Claudio Spiess, and Martin Hirzel. 2024. PDL: A Declarative Prompt Programming Language. (10 2024). doi:10.48550/arXiv.2410.19135

[35] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. 2023. Verifying the Verifier: eBPF Range Analysis Verification. In *Computer Aided Verification: 35th International Conference, CAV 2023, Paris, France, July 17–22, 2023, Proceedings, Part III* (Paris, France). Springer-Verlag, Berlin, Heidelberg, 226–251. doi:10.1007/978-3-031-37709-9_12