RIVA: Leveraging LLM Agents for Reliable Configuration Drift Detection

Sami Abuzakuk
EPFL
Lausanne, Switzerland
sami.abuzakuk@epfl.ch

Lucas Crijns
CYD Campus
armasuisse W+T
Lausanne, Switzerland
lucas.crijns@ar.admin.ch

Anne-Marie Kermarrec
EPFL
Lausanne, Switzerland
anne-marie.kermarrec@epfl.ch

Rafael Pires
EPFL
Lausanne, Switzerland
rafael.pires@epfl.ch

Martijn de Vos
EPFL
Lausanne, Switzerland
martijn.devos@epfl.ch

Abstract

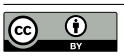
Infrastructure as code (IaC) tools automate cloud provisioning but verifying that deployed systems remain consistent with the IaC specifications remains challenging. Such *configuration drift* occurs because of bugs in the IaC specification, manual changes, or system updates. Large language model (LLM)-based agentic AI systems can automate the analysis of large volumes of telemetry data, making them suitable for the detection of configuration drift. However, existing agentic systems implicitly assume that the tools they invoke always return correct outputs, making them vulnerable to erroneous tool responses. Since agents cannot distinguish whether an anomalous tool output reflects a real infrastructure problem or a broken tool, such errors may cause missed drift or false alarms, reducing reliability precisely when it is most needed. We introduce RIVA (Robust Infrastructure by Verification Agents), a novel multi-agent system that performs robust IaC verification even when tools produce incorrect or misleading outputs. RIVA employs two specialized agents, a verifier agent and a tool generation agent, that collaborate through iterative cross-validation, multi-perspective verification, and tool call history tracking. Evaluation on the AIOpsLab benchmark demonstrates that RIVA, in the presence of erroneous tool responses, recovers task accuracy from 27.3% when using a baseline ReAct agent to 50.0% on average. RIVA also improves task accuracy 28% to 43.8% without erroneous tool responses. Our results show that cross-validation of diverse tool calls enables more reliable autonomous infrastructure verification in production cloud environments.

CCS Concepts

• **Computing methodologies** → **Artificial intelligence**.

Keywords

large language models, LLM agents, configuration drift, multi-agent systems, AIOps



EuroMLSys '26, Edinburgh, Scotland UK

ACM ISBN 979-8-4007-2605-7/2026/04
https://doi.org/10.1145/3805621.3807644

ACM Reference Format:
Sami Abuzakuk, Lucas Crijns, Anne-Marie Kermarrec, Rafael Pires, and Martijn de Vos. 2026. RIVA: Leveraging LLM Agents for Reliable Configuration Drift Detection. In *Sixth European Workshop on Machine Learning and Systems (EuroMLSys '26), April 27–30, 2026, Edinburgh, Scotland UK*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3805621.3807644

1 Introduction

IaC tools have become ubiquitous in modern IT operations, enabling automated and scalable infrastructure management [9]. Popular IaC tools such as Terraform [5], Ansible [16], and AWS CloudFormation [2] enable organizations to define, provision, and maintain their infrastructure through version-controlled configuration files, significantly reducing manual effort and configuration errors [7, 11, 15]. Instead of manually configuring cloud resources, IaC engineers describe the desired system state in declarative templates or scripts. These definitions are then processed by the IaC tool, which compares the declared state with the current environment and automatically applies the necessary changes to reach the target state. This approach ensures consistency across different environments (*e.g.*, development or production) and makes infrastructure reproducible, testable, and traceable [24]. The IaC paradigm is used by major companies such as Amazon, Netflix, Google and Facebook [9].

Despite these advantages, verifying that deployed infrastructure adheres to its specification remains a critical challenge [26]. The main problem is that during or after provisioning, the infrastructure may diverge from the intended state, *i.e.*, configuration drift [12]. During provisioning, subtle bugs in IaC modules, provider plugins, or cloud APIs can leave resources in a partially created or an inconsistent state, such as virtual machines being provisioned without the correct security groups or permissions only being applied to a subset of expected services [4]. Such bugs can propagate at scale, potentially leading to wide-scale service disruptions or vulnerabilities over time [14, 23]. Post-deployment, configuration drift introduces further complexities: automated software updates, manual changes by engineers that bypass IaC (*e.g.*, in response to an emergency), or performance tuning can modify resource properties in ways that violate the IaC specifications [29]. Detecting configuration drift requires continuous monitoring and correlating massive volumes of logs, events, and state changes to determine whether the live

environment still matches the intention of the IaC definitions. This verification effort is extremely labor-intensive and error-prone [23].

Recent advances in LLM-based agentic systems have shown strong potential to improve cloud reliability [20, 28]. In the context of IaC and configuration drift, agentic AI systems can interpret logs, events, and configuration states, summarize large volumes of telemetry data, and autonomously decide when and how to investigate anomalies [23]. By leveraging LLMs' abilities in pattern recognition, reasoning, and cross-correlating information from different sources, these agents can detect configuration drifts that are difficult to detect with traditional rule-based tools. Unlike rule-based approaches, which rely on predefined patterns and static thresholds, LLM-based agents can generalize across heterogeneous telemetry signals and adapt to emerging or previously unseen drift patterns without requiring manual rule updates [29]. This dynamic nature makes them particularly suited to complex cloud environments where the space of possible misconfigurations continuously evolves. Moreover, agentic workflows can iteratively refine hypotheses, query cloud APIs, and validate suspected issues against the IaC specification. This shows high potential to reduces manual effort, shorten incident-detection times, and to provide operators with actionable, high-level summaries of the configuration drift.

A key vulnerability of agentic AI in the context of IaC, however, is their implicit assumption that the tools they invoke always return correct and trustworthy outputs. In practice, IaC-related tools may fail or behave inconsistently due to API outages, throttling, transient network issues, parsing errors, or outdated provider implementations. Crucially, such erroneous tool responses are sometimes indistinguishable from genuine infrastructure misconfigurations: both manifest as unexpected states, missing fields, or contradictory information. As a result, an agent that relies solely on tool feedback cannot determine whether it is observing a real configuration drift or merely a faulty tool invocation. In Figure 1 we show the impact of a single erroneous tool on the success rate of localization and detection tasks in the AIOpsLab benchmark [3]. The presence of erroneous tools, silently returning incorrect information, negatively impact the task success rate, from 22.2% to 11.1% and 50.0% to 40.0% for the localization and detection tasks, respectively. Existing agentic solutions lack mechanisms for detecting or reasoning about these inaccuracies in tool outputs.

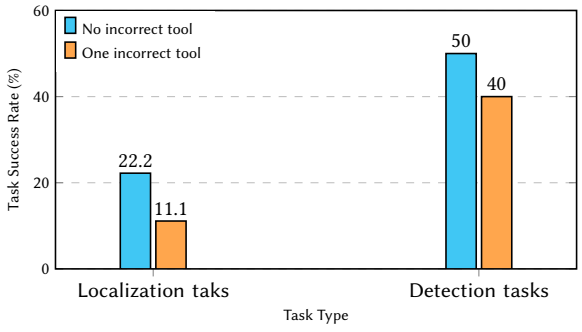


Figure 1: The impact of erroneous tools on the accuracy of error detection and localization tasks in the AIOpsLab benchmark by agentic AI systems.

This work introduces RIVA (Robust Infrastructure by Verification Agents), a novel multi-agent system designed to perform robust IaC-based infrastructure verification even when the tools it relies on produce incorrect or misleading outputs. The main insight of RIVA is that it is unlikely that two tool calls with the same goal will both be erroneous. Our system enables agents with the ability to detect, reason about, and overcome unreliable tool responses through iterative cross-validation, multi-perspective verification, and history tracking of tool results. At the core of RIVA are two specialized agents, a verifier and tool generation agent. Together, these agents converges on reliable conclusions about infrastructure compliance. Moreover, these agents have access to a tool call history which they use to generate and invoke unique tool calls to verify a single property in the IaC specifications.

We implement and evaluate our system on the AIOpsLab benchmark. We compare the performance of RIVA against that of a ReAct agent, which is a single agent that alternates between reasoning and tool calls. In the presence of unreliable tools, RIVA across all tasks recovers the task success rate from 27.3% when using a baseline ReAct agent to 50.0%. Furthermore, our results show superior accuracy and efficiency across all tasks even when tools generate correct responses, improving the average task success rate from 28% to 43.8%. Overall, we find that RIVA is an effective solution to mitigate the effect of incorrect tools and provide robust infrastructure verification.

Contributions. Our contributions are as follows:

- We design RIVA, a novel multi-agent system that is able to detect configuration drifts in the presence of erroneous tool call outputs (Section 3).
- We implement RIVA and integrate our system into AIOpsLab, an open-source framework for the structured evaluation of AI agents (Section 4).
- We evaluate the effectiveness of RIVA and show that it significantly outperforms a ReAct agent, in terms of task success rate, on tasks in the AIOpsLab benchmark, both with and without erroneous tool responses (Section 5). This comes with a manageable overhead in number of tokens used.

2 Background and Problem Description

We next provide background on IaC, configuration drift and agentic AI. We then formulate the main research challenge that this work addresses.

2.1 IaC and Configuration Drift

IaC tools such as Terraform, Ansible, and AWS CloudFormation allow operators to define the intended state of their infrastructure declaratively, enabling reproducibility, automation, and version control. In practice, however, deployed resources frequently diverge from the IaC definition—a phenomenon known as *configuration drift* [12]. Drift can arise during provisioning when subtle bugs or provider API inconsistencies cause resources to be created with missing or incorrect attributes without any explicit error being reported, or after provisioning when manual hotfixes, automated scaling actions, or emergency interventions modify parameters outside the IaC workflow. Over time, these discrepancies accumulate, making the operational state increasingly disconnected from

its specification, and detecting them at cloud scale requires extensive tooling and automation. A concrete Terraform example is provided in Appendix A.

2.2 Agentic AI

LLMs have recently emerged as a compelling solution for cloud engineering and improved reliability [13, 20, 27]. In particular, LLMs are able to process the heterogeneous and unstructured telemetry data that are common in modern cloud environments [17]. Their strengths in pattern recognition, summarization, and contextual reasoning make them well suited for interpreting complex system behavior and identifying potential issues, *e.g.*, the detection of concept drift. However, using an LLM in isolation is not sufficient: effective infrastructure verification requires iterative reasoning and continuous interaction with the live environment to fetch the appropriate data from nodes and services.

To enable such interaction, there is a shift towards adopting *agentic AI* where autonomous agents coordinate with external tools such as cloud APIs, telemetry endpoints, or custom scripts, to gather evidence and refine their understanding of the system state [6, 10]. More specifically, tools are external service interfaces that agents can programmatically invoke to execute functions beyond the inherent capabilities of the underlying LLMs, thus extending the capabilities of LLMs by providing direct programmatic access to telemetry and configuration data. A central part of the LLM agent's decision-making process is the selection, invocation, and orchestration of the available tools [18].

In the context of IaC and configuration drift, tool calls allow agents to probe the environment, query the properties of deployed resources, or retrieve telemetry needed to confirm whether the actual system matches the IaC specification. A detailed verification workflow is illustrated in Appendix C. A widely used approach for structuring such interactions is the ReAct framework, in which an agent alternates between natural-language reasoning steps and concrete tool invocations [30]. We refer to the resulting sequence of reasoning and tool-use decisions as a *trajectory*. In summary, agentic AI provides a promising direction for addressing the challenges posed by configuration drift.

2.3 Problem Description

The effectiveness of agentic AI in the context of IaC verification fundamentally depends on the reliability of the tools they invoke. When tools provide incomplete, stale, or misleading outputs, agents may incorrectly conclude that a resource is misconfigured or, conversely, that the infrastructure is healthy even when configuration drift has occurred. This motivates the need for *robust agentic frameworks* that are capable of successful operation even when the correctness of tool outputs cannot be guaranteed.

An illustrative example of how erroneous tool outputs can silently mislead an agent can be found in Appendix B.

In summary, the core problem is that agentic AI systems rely on tool outputs that may themselves be unreliable, incomplete, or stale, and current architectures provide no principled mechanism to detect or reason about such inconsistencies. As a result, agents cannot reliably distinguish genuine configuration drift from faulty tool behavior. This leads to the central research question of this work: *How can we design agentic AI systems that robustly verify IaC-defined infrastructure even when the tools they depend on return incorrect or misleading outputs?*

3 Design of RIVA

We design RIVA, a multi-agent architecture for IaC verification in the presence of unreliable tools. We visualize the RIVA system architecture and workflow in Figure 2. At the core of our system architecture are two collaborating LLM agents: a *Verifier Agent* and a *Tool Generation Agent*. The verifier agent has access to the IaC specification and finds properties to verify. The tool generation agent generates tool calls, executes these tool calls, collects the results, and sends the result back to the verifier agent.

The main insight behind RIVA is that even though a single tool call may silently return incorrect or misleading information, it is highly unlikely that multiple independent tools will all fail in the same way. In particular, tools are usually implemented as wrappers around complex commands in order to simplify the interaction with the agent. For example, in a Kubernetes environment, one might implement a `get_logs` tool using the Kubernetes executable, abstracting away the complexity of using this binary. However, accessing the logs of a pod can also be done by directly accessing the log files in the pod (i.e., with SSH). A subtle drift in the infrastructure configuration might render the original `get_logs` invalid but would probably not impact a direct access to the machine.

Thus, instead of trusting individual tool responses, we verify each property through cross-validation across diverse tool calls. By comparing their outputs, identifying contradictions, and iteratively refining verification attempts, an agent can distinguish genuine configuration drift from erroneous tool behavior. Therefore, both the verifier agent and tool generation agent have access to a shared data structure called the Tool Call History to verify infrastructure compliance with specifications.

In the following, we first elaborate on each component in our system and then describe the full RIVA workflow.

3.1 Tool History

The Tool History serves as the central data structure enabling agent collaboration and tool reliability assessment. It is implemented as a map where each key corresponds to a property identifier (a specific line in the infrastructure specification), and each value is an array containing up to K tool execution records. Every execution record is required to use a different tool. For example, with $K = 2$ and testing the reachability of node 1, the tool generation agent could add an execution record for `ping_node(id=1)` and `send_message(id=1)`. Effectively, the hyperparameter K defines the number of diagnostic paths that must be executed for the Verifier Agent to reach a conclusion. Once K records are available for a property, the Verifier Agent determines its compliance status by majority vote over the collected results; if no majority exists, the property is left unresolved and cannot be proven or disproven. We evaluate the impact of this parameter on the task success rate in Section 5.3.

Each record in the tool history contains three elements: (1) the executed command with its arguments, (2) the results generated by the tool, and (3) a brief analysis explaining what the results indicate about the property. This structure allows the Verifier Agent

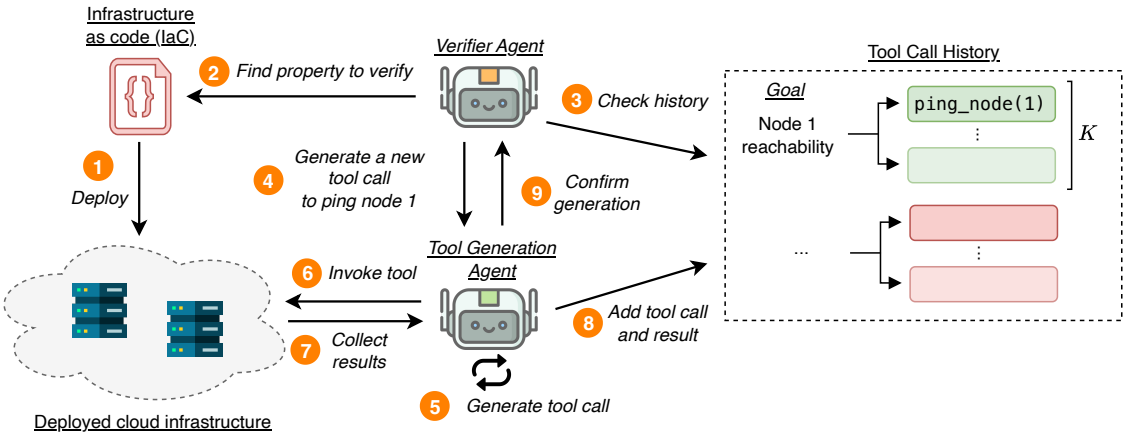


Figure 2: The RIVA system architecture and workflow.

to examine multiple verification attempts for the same property, facilitating the detection of inconsistent or unreliable tool outputs through cross-validation.

3.2 Tool Generation Agent

The Tool Generation Agent selects and instantiates verification commands from the available toolset for a specific property. When invoked, the agent first examines the Tool History to identify all previously executed tool calls for the target property. It then selects a tool not yet used for that property and determines a meaningful instantiation of it, ensuring diversity across verification attempts without relying on tools already present in the history. After instantiating the command, the agent executes it on the deployed infrastructure and collects the results. If the tool call is incorrect (wrong arguments, syntax errors, etc.), the agent corrects the issue and executes the command again. Finally, when the tool successfully runs on the infrastructure, the agent creates a new entry in the Tool History containing the command specification, the execution results, and an analysis that interprets what these results reveal about the property's compliance status.

3.3 Verifier Agent

The Verifier Agent is responsible for validating that the deployed infrastructure adheres to the system specification. For each property in the specification, the agent queries the Tool History to determine whether verification attempts exist for that property. If no entry is found, the agent requests the Tool Generation Agent to create an appropriate verification command. If entries exist, the agent analyzes the stored tool results to determine whether they provide sufficient evidence to conclude that the property is satisfied or violated. Crucially, the agent formulates a conclusion about the property only if K entries are available (i.e., if K distinct diagnostic paths were explored to generate results). While preventing the agent from reaching a conclusion too quickly, this design still allows the agent to validate other properties. In particular, because the tool generation agent generates one tool call at a time, the verifier agent has a reasoning step interleaved between each generation. This allows the agent to reevaluate its plan frequently and, for example, in situations where the initially selected property is not strictly necessary to move forward in the verification task, add a new goal to the tool history and abandon the first goal. Note that in such a case, because of the K constraint, the abandoned goal will not be considered as proven or disproven by the agent.

4 Experimental Setup

We now describe the experimental setup used to evaluate RIVA, including the AIOpsLab benchmark, the implementation details of RIVA, and the evaluation process.

4.1 The AIOpsLab Benchmark

We conduct our evaluation using AIOpsLab, an open-source framework developed by Microsoft Research for designing, developing, and evaluating autonomous AIOps agents [3]. AIOpsLab provides a holistic infrastructure that can deploy microservice cloud environments, inject faults, generate workloads, and export comprehensive telemetry data. Moreover, AIOpsLab supports several critical AIOps tasks including incident detection, localization, root cause diagnosis, and mitigation, making AIOpsLab an ideal testbed for evaluating infrastructure verification by LLM agents. In our setting, we focus exclusively on identifying infrastructure issues rather than resolving them, and therefore evaluate on all task types except mitigation tasks.

4.2 Baseline Agent

For comparison, we use the ReAct agent implementation in AIOpsLab as our baseline. The ReAct framework enables agents to interleave reasoning and action steps, allowing them to generate verbal reasoning traces before selecting and executing tools [30]. We select ReAct as the baseline because it heavily relies on tool invocations and is the standard pattern adopted by most existing agentic AI systems [18].

4.3 Tool Reliability Evaluation

To evaluate the effectiveness of our approach in handling incorrect tool outputs, we modify the behavior of some observability tools

provided by AIOpsLab. The AIOpsLab implementation exposes various tools to collect telemetry data from deployed infrastructure. Specifically, we configure the `get_logs` and `read_metrics` functions to return empty strings rather than the expected telemetry data. These modifications simulate tool unreliability while maintaining the distinction between incorrect outputs and genuine errors, tool invocations that violate the expected interface or encounter execution failures still raise exceptions as expected. This design choice reflects scenarios where tools execute successfully but return misleading or incomplete information due to configuration issues, API timeouts, or data collection failures. To ensure fair comparison between both implementations, the maximum number of steps is set to 45 for each, matching the upper bound defined by AIOpsLab. For RIVA, a multi-agent system, this limit represents the combined steps executed by both agents.

4.4 Evaluation Protocol

We evaluate both the baseline ReAct agent and our proposed system across all root cause analysis tasks available in AIOpsLab. The benchmark contains three types of tasks: *localization tasks* identify where the root cause of a particular fault lies, *detection tasks* aim to determine whether an incident has occurred, and *analysis tasks* diagnose the root cause of identified incidents. For each task category, we conduct experiments under different tool reliability conditions, systematically introducing incorrect behavior in one the two observability tools (`get_logs`, `read_metrics`). We run each task 5 times and measure the accuracy of both systems in correctly completing each task type. For all experiments, we use the model `gpt-oss:120b` [1] directly, without any fine-tuning.

5 Experimental Evaluation

We evaluate the performance of RIVA. Our evaluation answers the following questions:

- What is the task success rate of RIVA and the ReAct baseline agent on tasks in the AIOpsLab benchmark, both with and without erroneous tool responses (Section 5.1)?
- What is the step count and number of tokens used for RIVA and the ReAct baseline agent on tasks in the AIOpsLab benchmark (Section 5.2)?
- How does the tool call history size (parameterized by K) impact the performance of RIVA (Section 5.3)?

5.1 Task Success Rate of RIVA and ReAct agents

We run RIVA (with $K = 2$) and the ReAct agent on all detection, localization, and analysis tasks in the AIOpsLab benchmark. Figure 3 shows the task success rate of each task type with and without erroneous tools. In the presence of incorrect tools, the task success rate of RIVA is always greater than or equal to that of the ReAct agent with only correct tools. For instance, the accuracy of localization tasks with the ReAct implementation is 22.2% with correct tools and 11.1% with an incorrect `get_logs` tool implementation. RIVA achieves an accuracy of 20.0% with the same incorrect tool. Notably, across all tasks, RIVA improves the average accuracy from 28.0% to 43.8% even when all tool responses are correct. Careful analysis of the agentic trajectories reveals that most of this improvement stems from the multi-agent design of RIVA. By separating responsibilities, the verifier agent focuses exclusively on interpreting tool call results rather than managing their creation and correction. This division of labor reduces reasoning errors that plague ReAct agents, which often lose sight of their initial goal after saturating their context with failed tool executions.

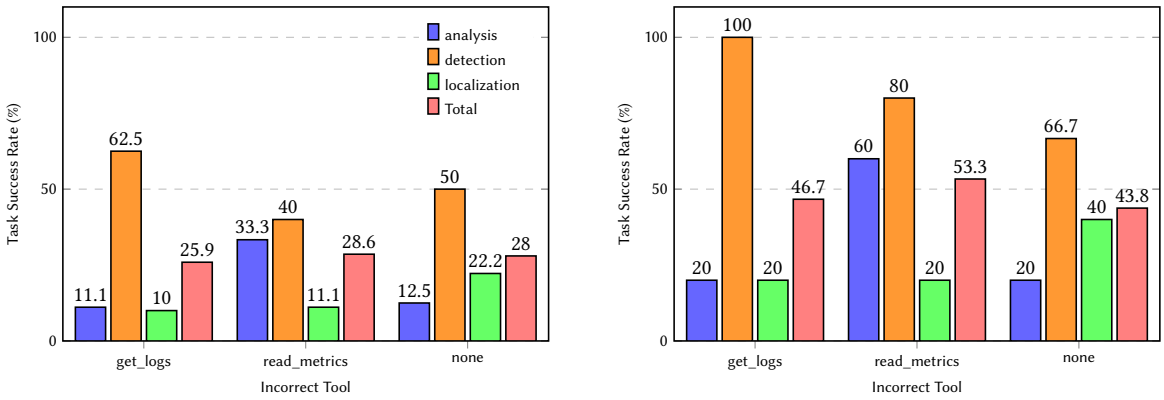
These results show that RIVA can reach superior task success rates even with erroneous tool responses.

However, RIVA fails to reach accuracies similar to those achieved when executed with correct tools. This suggests that our system does not fully mitigate the impact of erroneous tools. In particular, for localization tasks, RIVA reaches 40.0% task success rate when all tool calls are correct but achieves 20.0% when the `get_logs` and `read_metrics` tools are incorrect. An inspection of several trajectories reveals that the tool generation agent is responsible for this loss in accuracy. Specifically, because this agent generates tool calls, it is responsible for correcting incorrect tool calls. For example, if the agent fails to identify a correct service name, it will attempt to find the correct service name by itself using its tools. These tool calls are never added to the tool history, meaning that subsequent calls to the tool generator will likely have to redo the same work. This increases the number of steps needed for each generation and the probability of error, leading to decreased accuracy. This issue could be mitigated by dynamically appending exploratory tool calls to the history, enabling the tool generation agent to build upon previous correction attempts rather than repeating the same discovery process.

Notably, the presence of erroneous tool calls *sometimes improves the accuracy of the ReAct agent.* For example, the task success rate of the ReAct agent for detection tasks is 50.0% whereas the success rate with an incorrect `get_logs` reaches 62.5%. We note that detection tasks are the easiest tasks as they can be solved by only finding an irregular metric or log. Thus, by making `get_logs` return an empty string, the base agent ignores the results and immediately focuses on the metric results. Metrics being more focused and less verbose lead to fewer hallucinations and thus better results. Overall, the results in Figure 3 suggest that RIVA successfully leverages its design to mitigate the effect of incorrect tools.

5.2 Efficiency of RIVA and ReAct agents

We compare the efficiency of RIVA and ReAct by quantifying trajectory length and token usage, which directly capture the duration and computational cost of agentic reasoning. With correct tools, 80% of RIVA ($K = 2$) tasks complete within 15 steps versus 60% for ReAct; RIVA's step count peaks at 17 while 33% of ReAct runs hit the 45-step cap, and peak token usage is 38 000 for RIVA versus 78 000 for ReAct. We attribute this to the two-agent design of RIVA, which distributes context across agents, reducing hallucination risk and the number of corrective steps. With erroneous tools, RIVA still requires at most 17 steps whereas 37% of ReAct tasks exceed this threshold; peak token usage is 50 000 for RIVA versus 90 000 for ReAct. Detailed CDF distributions are shown in Appendix D.



(a) The task success rate of a ReAct agent.

(b) The task success rate of RIVA.

Figure 3: The task success rate of RIVA (with $K = 2$) and a ReAct agent in the presence of erroneous tools and without any erroneous tool.

Table 1 Average success rate of ReAct and RIVA with different K on AIOpsLab

Agent	Only correct tools	incorrect get_logs	incorrect read_metrics
ReAct	28.00	25.93	28.57
RIVA (K=1)	27.67	24.80	29.02
RIVA (K=2)	43.75	46.67	53.33
RIVA (K=3)	0	0	0

5.3 The Impact of K on Efficiency

We now examine the impact of tuning the hyperparameter K, which governs the number of unique tool calls necessary to reach a conclusion. The average success rate of ReAct and RIVA with K set to 1, 2 and 3 are shown in Table 1.

When K is set to 1, RIVA achieves similar numbers to ReAct. By using only 1 tool call to validate a property, the advantages of RIVA are effectively negated and it falls into the same trap as ReAct. When K is set to 2, as seen in previous subsections, RIVA outperforms ReAct and is able to avoid the accuracy degradation caused by the incorrect tool call. However, when K is higher than 2, the accuracy falls to 0%. This failure stems from a fundamental constraint of AIOpsLab: it does not offer enough diagnostic paths for the generator agent to produce three distinct tool calls for the same goal, causing it to loop until hitting the step limit. A detailed analysis of this failure mode is provided in Appendix E.

These results highlight that RIVA performance depends critically on hyperparameter K. If the system cannot generate at least K distinct tool calls to verify the same property, it will fail to reach a conclusion. This hyperparameter directly reflects the diagnostic flexibility afforded to the agent. In constrained environments like AIOpsLab, where the agent has limited alternative diagnostic paths, K should be set lower accordingly. This analysis also reveals a promising direction for improving RIVA design: when the Tool Generation agent cannot identify K distinct paths, the system could continue its analysis with reduced confidence levels rather than halting entirely.

6 Related Work

The challenge of ensuring robust agent execution in the presence of tool errors has received growing attention in recent research on LLM-based agentic systems. Approaches generally fall into two categories: handling explicit execution failures and mitigating subtle, silent errors.

Vuddanti et al. introduce PALADIN [25], a self-correcting language model agent designed to handle tool malfunctions such as timeouts and API exceptions. Their approach focuses on enabling agents to recover from tool execution failures through self-correction mechanisms. However, PALADIN assumes that tool failures manifest as explicit exceptions (execution errors).

Similarly, Sheffler proposes a temporal expression language for monitoring agent behavior and detecting deviations from expected behavioral patterns [19]. This approach monitors execution traces of tool calls and state transitions to identify anomalies in agent actions, such as improper tool sequencing and failed coordination. While effective for detecting behavioral regressions, this method requires predefined temporal patterns and does not explicitly address the problem of verifying tool output correctness.

As opposed to explicit failures, the challenge of silent errors, where a tool successfully executes but returns an incorrect output, has been highlighted as a critical issue in recent work [22]. Tools Fail investigates methods for LLMs to learn to doubt tools and detect these mistakes, often using in-context interventions like checklists. While this work confirms the existence and importance of the silent error problem, it primarily focuses on the LLM's capacity for detection based on internal expectations, whereas our work provides a structured, external mechanism for tool output verification.

Some work tackle the tool error problems by improving recovery. Several approaches explore learning-based error recovery mechanisms, often involving explicit reflection processes that diagnose failed tool calls and propose corrected alternatives [21]. Additionally, AgentDebug provides a debugging framework that isolates root-cause failures and generates corrective feedback, enabling agents to iteratively recover from errors [31].

On the side of provable safety, VeriGuard tackles a related challenge by integrating formal verification into the code generation process to ensure 'correct-by-construction' agent policies, using an iterative refinement loop guided by a verifier's counterexamples [8].

While these works address various aspects of tool reliability and error handling—from explicit exceptions (PALADIN) to internal detection of silent errors (Tools Fail) and policy-level verification (VeriGuard), they primarily focus on detecting and recovering from explicit tool failures or execution errors. Our approach differs by specifically targeting scenarios where tools execute without raising exceptions but return incorrect results, requiring the agent to reason about tool output reliability through cross-validation and iterative verification rather than relying on explicit error signals.

7 Conclusions

This paper presents RIVA, a multi-agent system that detects configuration drift in cloud infrastructure even when tools return incorrect data. By cross-validating multiple independent tool calls targeting the same property, RIVA distinguishes real infrastructure problems from faulty tool outputs. Evaluation on the AIOpsLab benchmark demonstrates that RIVA, in the presence of erroneous tool responses, recovers task accuracy from 27.3% when using a baseline ReAct agent to 50.0% on average. RIVA also improves task accuracy 28% to 43.8% without erroneous tool responses. Moreover, our system completes most tasks more efficiently, using fewer steps and tokens compared to a baseline ReAct agent. Thus, RIVA demonstrates that cross-validation enables more reliable autonomous infrastructure verification in production environments.

Future work should explore two promising directions to further enhance the robustness and applicability of RIVA. First, the hyperparameter K could be determined dynamically rather than fixed at deployment time. A self-modulating mechanism would allow the system to adjust the required number of diagnostic paths on a per-property basis, increasing K when sufficient alternative tool calls are available and reducing it in constrained environments, making the system more adaptive across diverse deployment settings. Second, our current evaluation simulates tool unreliability by having tools return empty strings, providing a controlled and tractable starting point. A natural extension would be to evaluate RIVA against partially valid tool outputs, where tools return plausible but subtly incorrect information, such as stale metrics or misattributed log entries. Such experiments would further stress-test the cross-validation mechanism and demonstrate RIVA's robustness in the silent error scenarios most commonly encountered in production cloud environments.

Acknowledgments

This work was partially supported by Cyber Defence Campus (project number AR-CYD-C-024, contract number 8010075905) and the Swiss National Science Foundation, under the project FRIDAY: Frugal, Privacy-Aware and Practical Decentralized Learning, SNSF proposal No. 10.001.796. We thank Gérôme Bovet, Sayan Biswas, and Milos Vujasinovic for their inputs and insightful discussions.

References

[1] Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K Arora, Yu Bai, Bowen Baker, Haiming Bao, et al. 2025. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925* (2025).

[2] Amazon Web Services. 2025. AWS CloudFormation. https://aws.amazon.com/cloudformation/. Accessed: 2025-02-24.

[3] Yinfang Chen, Manish Shetty, Gagan Somashekar, Minghua Ma, Yogesh Simmhan, Jonathan Mace, Chetan Bansal, Rujia Wang, and Saravan Rajmohan. 2025. Aiopslab: A holistic framework to evaluate ai agents for enabling autonomous clouds. *arXiv preprint arXiv:2501.06706* (2025).

[4] Georgios-Petros Drosos, Thodoris Sotiropoulos, Georgios Alexopoulos, Dimitris Mitropoulos, and Zhendong Su. 2024. When your infrastructure is a buggy program: Understanding faults in infrastructure as code ecosystems. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024), 2490–2520.

[5] HashiCorp. 2025. Terraform. https://www.terraform.io/. Accessed: 2025-02-24.

[6] Laurie Hughes, Yogesh K Dwivedi, Tegwen Malik, Mazen Shawosh, Mousa Ahmed Albashrawi, Il Jeon, Vincent Dutot, Mandanna Appanderanda, Tom Crick, Rahul De', et al. 2025. AI agents and agentic systems: A multi-expert analysis. *Journal of Computer Information Systems* (2025). doi:10.1080/08874417.2025.2483832

[7] Indika Kumara, Martín Garriga, Angel Urbano Romeu, Dario Di Nucci, Fabio Palomba, Damian Andrew Tamburri, and Willem-Jan van den Heuvel. 2021. The do's and don'ts of infrastructure code: A systematic gray literature review. *Information and Software Technology* 137 (2021), 106593.

[8] Lesly Miculicich, Mihir Parmar, Hamid Palangi, Krishnamurthy Dj Dvijotham, Mirko Montanari, Tomas Pfister, and Long T Le. 2025. VeriGuard: Enhancing LLM Agent Safety via Verified Code Generation. *arXiv preprint arXiv:2510.05156* (2025).

[9] Kief Morris. 2020. *Infrastructure as code.* O'Reilly Media.

[10] San Murugesan. 2025. The Rise of Agentic AI: Implications, Concerns, and the Path Forward. *IEEE Intelligent Systems* 40, 2 (2025). doi:10.1109/MIS.2025.3544940

[11] Ruben Opdebeeck, Bram Adams, and Coen De Roover. 2025. Analysing Software Supply Chains of Infrastructure as Code: Extraction of Ansible Plugin Dependencies. In *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 181–192.

[12] Claus Pahl, Niyazi Gokberk Gunduz, Ovgüm Can Sezen, Ali Ghamgosar, and Nabil El Ioini. 2025. Infrastructure as Code–Technology Review and Research Challenges. (2025).

[13] Kannan Parthasarathy, Karthik Vaidhyanathan, Rudra Dhar, Venkat Krishnamachari, Adyansh Kakran, Sreemaee Akshathala, Shrikara Arun, Amey Karan, Basil Muhammed, Sumant Dubey, et al. 2025. Engineering LLM Powered Multi-Agent Framework for Autonomous CloudOps. In *2025 IEEE/ACM 4th International Conference on AI Engineering–Software Engineering for AI (CAIN)*. IEEE, 201–211.

[14] Akond Rahman, Sarah Elder, Faysal Hossain Shezan, Vanessa Frost, Jonathan Stallings, and Laurie Williams. 2018. Bugs in infrastructure as code. *arXiv preprint arXiv:1809.07937* (2018).

[15] Akond Rahman, Rezvan Mahdavi-Hezaveh, and Laurie Williams. 2019. A systematic mapping study of infrastructure as code research. *Information and Software Technology* 108 (2019), 65–77.

[16] Red Hat. 2025. Ansible. https://www.ansible.com/. Accessed: 2025-02-24.

[17] Devjeet Roy, Xuchao Zhang, Rashi Bhave, Chetan Bansal, Pedro Las-Casas, Rodrigo Fonseca, and Saravan Rajmohan. 2024. Exploring LLM-Based Agents for Root Cause Analysis. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering* (Porto de Galinhas, Brazil) *(FSE 2024)*. doi:10.1145/3663529.3663841

[18] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems* 36 (2023), 68539–68551.

[19] Thomas J Sheffler. 2025. An Approach to Checking Correctness for Agentic Systems. *arXiv preprint arXiv:2509.20364* (2025).

[20] Manish Shetty, Yinfang Chen, Gagan Somashekar, Minghua Ma, Yogesh Simmhan, Xuchao Zhang, Jonathan Mace, Dax Vandevoorde, Pedro Las-Casas, Shachee Mishra Gupta, et al. 2024. Building ai agents for autonomous clouds: Challenges and design principles. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*. 99–110.

[21] Junhao Su, Yuanliang Wan, Junwei Yang, Hengyu Shi, Tianyang Han, Junfeng Luo, and Yurui Qiu. 2025. Failure Makes the Agent Stronger: Enhancing Accuracy through Structured Reflection for Reliable Tool Interactions. *arXiv preprint arXiv:2509.18847* (2025).

[22] Jimin Sun, So Yeon Min, Yingshan Chang, and Yonatan Bisk. 2024. Tools Fail: Detecting Silent Errors in Faulty Tools. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (Eds.). Association for Computational Linguistics, Miami, Florida, USA, 14272–14289. doi:10.18653/v1/2024.emnlp-main.790

[23] Gogulakrishnan Thiyagarajan, Vinay Bist, and Prabhudarshi Nayak. 2024. AI-Driven Configuration Drift Detection in Cloud Environments. *Gogulakrishnan Thiyagarajan, Vinay Bist, Prabhudarshi Nayak.(2024). AI-Driven Configuration Drift Detection in Cloud Environments. International Journal of Communication Networks and Information Security (IJCNIS)* 16, 5 (2024), 721–743.

[24] Alexandre Verdet. 2023. *Exploring security practices in infrastructure as code: An empirical study.* Ecole Polytechnique, Montreal (Canada).

[25] Sri Vatsa Vuddanti, Aarav Shah, Satwik Kumar Chittiprolu, Tony Song, Sunishchal Dev, Kevin Zhu, and Maheep Chaudhary. 2025. PALADIN: Self-Correcting Language Model Agents to Cure Tool-Failure Cases. *arXiv preprint arXiv:2509.25238* (2025).

[26] Rosemary Wang. 2022. *Infrastructure as Code, Patterns and Practices: With Examples in Python and Terraform.* Simon and Schuster.

[27] Yiming Xiang, Zhenning Yang, Jingjia Peng, Hermann Bauer, Patrick Tser Jern Kon, Yiming Qiu, and Ang Chen. 2025. Automated bug discovery in cloud infrastructure-as-code updates with llm agents. In *2025 IEEE/ACM International Workshop on Cloud Intelligence & AIOps (AIOps)*. IEEE, 20–25.

[28] Zhenning Yang, Archit Bhatnagar, Yiming Qiu, Tongyuan Miao, Patrick Tser Jern Kon, Yunming Xiao, Yibo Huang, Martin Casado, and Ang Chen. 2025. Cloud infrastructure management in the age of ai agents. *ACM SIGOPS Operating Systems Review* 59, 1 (2025), 1–8.

[29] Zhenning Yang, Hui Guan, Victor Nicolet, Brandon Paulsen, Joey Dodds, Daniel Kroening, and Ang Chen. 2025. Automated Cloud Infrastructure-as-Code Reconciliation with AI Agents. *arXiv preprint arXiv:2510.20211* (2025).

[30] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*. arXiv:2210.03629 https://openreview.net/pdf?id=WE_vluYUL-X

[31] Kunlun Zhu, Zijia Liu, Bingxuan Li, Muxin Tian, Yingxuan Yang, Jiaxun Zhang, Pengrui Han, Qipeng Xie, Fuyang Cui, Weijia Zhang, et al. 2025. Where LLM Agents Fail and How They can Learn From Failures. *arXiv preprint arXiv:2509.25370* (2025).

A Terraform Configuration Example

This appendix provides a concrete Terraform configuration snippet illustrating how IaC definitions can diverge from live infrastructure, along with representative examples of configuration drift arising during and after provisioning.

Listing 1 A Terraform configuration snippet provisioning a web server and assigning it a security group.

```
resource "aws_instance" "web" {
  ami           = "ami-123"
  instance_type = "t2.micro"

  vpc_security_group_ids = [aws_security_group.web_sg.id]

  tags = {
    Name        = "web-server"
    Environment = "production"
  }
}
```

The snippet in Listing 1 declaratively defines a web server instance, including its operating system image, compute type, and network security group. It also attaches identifying tags, which are commonly used to organize resources and express their intended role within the infrastructure. This approach enables reproducibility, automation, version control, and consistency.

In practice, deployed resources frequently diverge from the IaC definition, which is known as *configuration drift* [12]. Configuration drift can happen during provisioning, for instance when subtle bugs in IaC modules or provider APIs cause resources to be created with missing or inconsistent attributes. For instance, an intermittent cloud API issue may result in the instance in Listing 1 being provisioned without the intended security group due to a silent error in the backend, even though Terraform reports success. Such mismatches are difficult to detect immediately because the IaC tool may not report the underlying inconsistency.

Configuration drift can also occur after provisioning. Manual hotfixes, automated scaling actions, system updates, or emergency interventions may modify configuration parameters outside the IaC workflow. An operator responding to an incident might manually update the instance's security group or temporarily open an SSH port. Due to human error, these changes can remain in the live environment but are not captured in the Terraform file. Over time, these discrepancies accumulate, making the operational state increasingly disconnected from its specification.

B Tool Reliability Illustrative Example

This appendix provides a worked example of how erroneous tool outputs can silently mislead an agent, motivating the reliability problem addressed by RIVA.

Illustrating example. Consider Listing 2 with a `ping_node` function call that pings a node based on its identifier. Each node identifier is mapped to an expected IP address and the agent verifies reachability of a node by invoking `ping_node(id)`. Under normal circumstances, this tool call returns correct information about the intended node. However, drift or network reconfiguration may silently invalidate these assumptions. For instance, the router might

Listing 2 A simplified tool used by an agent to verify node reachability based on an expected IP mapping.

```
# Expected mapping between logical node identifiers and
↪ their IP addresses
nodes = {
    "0": "172.17.0.5",
    "1": "172.17.0.6",
}

# Tool invoked by the agent to check whether a node is
↪ reachable
def ping_node(node_id):
    return ping(nodes[node_id])
```

reassign IP addresses after a network event, or an operator may manually update network settings during an emergency, invalidating the IP-to-node mapping. Because the tool returns syntactically correct output, the agent has no direct signal that anything is wrong. For example, it concludes that node 1 is healthy even though the ping was forwarded to an unrelated device.

While the above example is straightforward, the underlying issue becomes far more severe in realistic cloud environments. Modern agentic systems interact with dozens of heterogeneous tools, each with their own failure modes, silent inconsistencies, and assumptions. As the number of tools and unknowns grows, the inability of the agent to distinguish genuine configuration drift from misleading but syntactically valid tool outputs increases.

C IaC Verification Workflow

This appendix illustrates the end-to-end workflow of an LLM agent verifying IaC-defined infrastructure, as referenced in Section 2.

An engineer first deploys some IaC specification (step 1). Then, an LLM agent collects data by using its available tools (steps 2 and 3), and reasons about the data (step 4) to identify potential issues. This process may repeat for multiple steps. Once the agent completes its task, it provides a report to the engineer who can further investigate identified issues.

D Detailed Efficiency Analysis

This appendix presents the full CDF distributions for step count and token usage of RIVA and the ReAct baseline referenced in Section 5.2.

Figure 5 (right) shows that when all tool responses are correct, 80% of all tasks are completed in 15 steps with RIVA (and $K = 2$) while only 60% of ReAct runs complete within this number of steps. Moreover, the maximum number of steps with RIVA ($K = 2$) is 17 while 33% of the runs with the ReAct agent require 45 steps, the maximum number of steps allowed before the task automatically aborts. Similarly, Figure 6 (right) shows that the distribution of maximum number of tokens used per task for RIVA ($K = 2$) is within the range of the ReAct agent, although the maximum number of tokens for RIVA ($K = 2$) is 38 000 tokens, compared to ReAct where a task requires at most 78 000 tokens. We attribute this to the two-agent design of RIVA, which effectively distributes the information between the two agents, resulting in smaller context usage, reducing the maximum token count and lowering the chances of

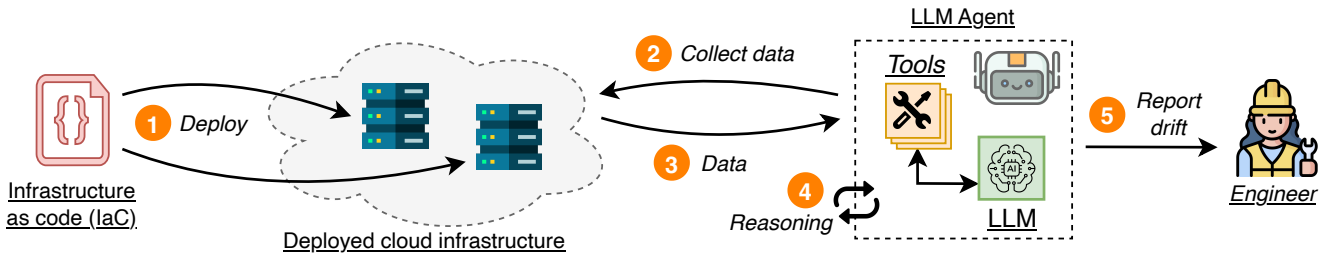


Figure 4: The workflow of configuration drift detection by LLM agents on infrastructure deployed by IaC.

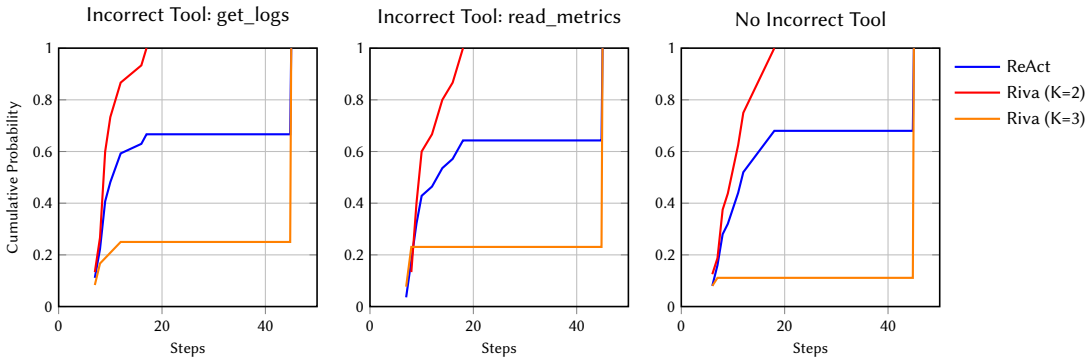


Figure 5: The distribution of number of steps required to complete tasks, for RIVA and the ReAct agent, with and without erroneous tool responses.

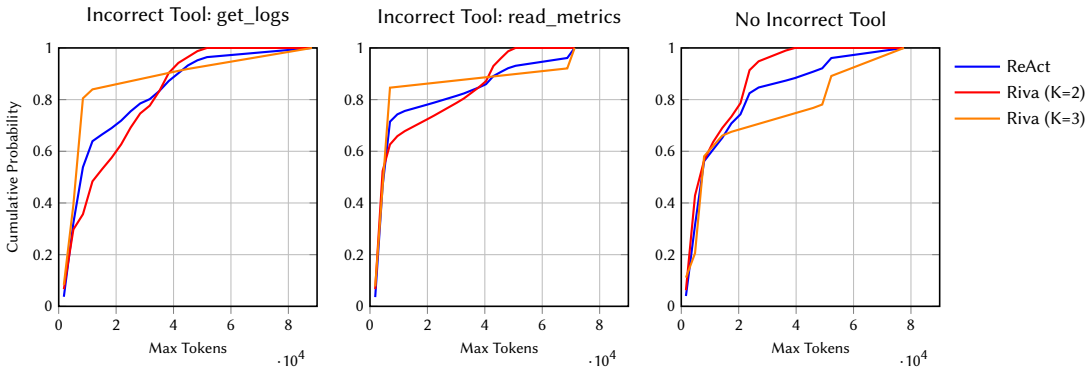


Figure 6: The distribution of maximum number of tokens required to complete tasks, for RIVA and the ReAct agent, with and without erroneous tool responses.

hallucination. The smaller context sizes in RIVA also reduce the number of incorrect tool calls per step, decreasing the amount of corrective steps the agent takes to fix tool calls.

In the presence of incorrect tools, RIVA uses fewer steps than ReAct on average to finish a task (see Figure 5 left and middle). In particular, RIVA requires at most 17 steps with erroneous tools whereas 37% of all ReAct tasks requires more than 17 steps. However, Figure 6 (left and middle) shows that RIVA uses a higher maximum number of tokens per task when tools are incorrect, though the tendency is inverted after 40000 tokens as the top 20% of ReAct runs exceed the top 20% of RIVA runs. Similarly to the experiment with only correct tools, the maximum value is much smaller than ReAct: 50 000 tokens for RIVA versus 90 000 for ReAct. Upon further inspection, RIVA starts generating more commands when erroneous tools are present compared to when all tools provide correct outputs, resulting in higher token usage that allows the system to avoid the negative effects of incorrect tools. ReAct, on the other hand, is not able to recover from incorrect results and exhausts all of its steps trying to reason over them. Overall, the design of RIVA allows for more effective use of its resources compared to the ReAct agent, particularly in the maximum number of tokens used to solve tasks.

E K Hyperparameter Sensitivity Analysis

This appendix provides a detailed analysis of why RIVA fails completely when $K = 3$, as referenced in Section 5.3.

Figure 5 and Figure 6 show that $K = 3$ leads to significantly different results: 83% of all tasks use fewer than 15000 tokens while 77% of all runs reach 45 steps, the maximum allowed in our evaluation. Deeper analysis reveals that most executions fail at generating a third command. In particular, AIOpsLab does not offer enough flexibility to the generator agent to produce more than 2 different tool calls for a specific goal. As soon as the verifier agent tasks the generator agent to generate a third command, the generator continues trying until it reaches the maximum number of steps. Moreover, as all generated tool calls are incorrect, they only produce a small number of tokens (an error message from the benchmark), explaining the small maximum token values observed.