

Date of submission July 5, 2026. Publication and current-version dates to be assigned by IEEE Access.

Digital Object Identifier To be assigned by IEEE Access

Evaluating Self-Reflective LLM Agents for Autonomous Kubernetes Troubleshooting

MUSTAFA MERT SÜERKAN¹, EMIN KUGU^{2,3} (Member, IEEE), and HAKAN EMEKCI⁴ (Member, IEEE)

¹TED University, Graduate School, 06420, Ankara, Türkiye

²TED University, Software Engineering Department, 06420, Ankara, Türkiye

³Universitat Politècnica de València, Telecommunication and Multimedia Applications University Research Institute (iTEAM), València, 46730, Spain

⁴Hacettepe University Institute of Informatics, Ankara, Türkiye

Corresponding author: Emin Kugu (emin.kugu@tedu.edu.tr, ekugu@upvnet.upv.es).

ABSTRACT Kubernetes is central to cloud-native deployment, yet resolving cluster incidents still demands considerable operational knowledge, and the dependability of large language model (LLM) agents for autonomous remediation is unclear. This paper examines whether Reflexion, a verbal reinforcement learning method driven by self-produced failure analysis, improves LLM agents on Kubernetes troubleshooting tasks. We built a Reflexion-based remediation system with a deterministic programmatic evaluator and assessed it in 1,500 controlled runs over five Claude models, ten fault scenarios, and three configurations (one-shot baseline, two-try without reflection, and full Reflexion with at most one reflection per run). Permitting a second attempt raised overall success from 38.6% to 68.4%, whereas adding Reflexion increased it only to 69.2%. Reflexion slightly reduced single-fault success (-2.0 percentage points) but raised it by 12.0 points in both multi-fault cases; with only one double-fault and one triple-fault scenario tested, this result is hypothesis-generating and needs validation on a larger benchmark. Two post-hoc controls on the three currently available models qualify these results. Forwarding the evaluator's verbatim failure reason without reflection outperformed full Reflexion on identically instantiated multi-fault scenarios (76.4% versus 26.0%), so most of the multi-fault gain comes from the external failure signal. Allowing the actor to verify its own fix within one attempt recovered about half of the retry effect on single-fault scenarios (67.5% to 80.8%), while the externally verified second attempt remained superior (92.5%). Mid-capability models benefited most and offered the best cost-latency trade-off, weaker models did not use reflection effectively, and the strongest model showed a ceiling effect. The results support selective use of self-reflection for complex or unresolved incidents rather than routine use for every failure.

INDEX TERMS AIOps, autonomous operations, DevOps, Kubernetes, large language model agents, Reflexion, self-reflection, incident remediation.

I. INTRODUCTION

Modern cloud-native systems increasingly depend on Kubernetes to manage containerized workloads over distributed infrastructure. Although Kubernetes offers declarative deployment, service discovery, scaling, and self-healing mechanisms, its operating model also makes production incident diagnosis more difficult. A fault can stem from a pod, deployment, service, ConfigMap, probe, port mapping, image definition, or resource constraint, while the visible symptom may be separated from the actual root cause. In typical practice, operators examine cluster state, events, logs, and YAML manifests before making changes with `kubectl` commands or infrastructure-as-code workflows. This process demands

specialized expertise and has a direct effect on mean time to recovery (MTTR). Empirical studies of large-scale cloud services underline the weight of this burden: most production incidents are mitigated not by code changes but by configuration fixes, rollbacks, and infrastructure-level interventions, which makes misconfiguration repair a first-order operational concern [1].

Native Kubernetes tools, including the Horizontal Pod Autoscaler and Vertical Pod Autoscaler, automate selected scaling decisions but do not offer broad diagnosis and repair for arbitrary configuration errors. Monitoring stacks such as Prometheus and Grafana increase observability, yet they typically do not perform autonomous remediation. LLM-

based agents can inspect cluster state, call tools, reason about symptoms, and apply fixes, which raises a reliability question: does an agent learn from a failed remediation attempt, or does self-reflection mainly add latency and cost without improving the outcome?

Reflection in LLM agents refers to an explicit post-hoc reasoning step in which the agent reviews its own prior output, trajectory, or failure and converts that review into guidance for a later attempt. It is related to broader self-improvement methods such as Self-Refine, where a model critiques and revises its own answer, and Tree of Thoughts, where alternative reasoning paths are explored before selecting a solution [2], [3]. Reflexion formalizes this idea as verbal reinforcement learning: instead of updating model parameters, the agent produces a natural-language reflection after an unsuccessful trajectory, stores it as episodic memory, and conditions the next trial on that memory [4]. In agentic systems, this mechanism is especially important because the model does not merely generate text; it selects tools, observes external feedback, and changes its plan across multiple interaction steps [5], [6]. Prior work on generative agents also shows that memory and reflection can help agents synthesize past observations into higher-level behavioral guidance, suggesting that reflection can function as a lightweight substitute for experience when retraining is not feasible [7].

We therefore treat reflection as a structured failure-analysis step, not as a generic second thought. A useful reflection identifies which hypothesis was wrong, which evidence was overlooked, which action partially changed the system state, and which repair to attempt next. This distinction matters for Kubernetes remediation because failures are interactive and stateful: an agent may fix an image tag but leave a service targetPort mismatch, repair a ConfigMap reference but still face a probe error, or misinterpret a symptom that appears far from the underlying resource. Reflection may help in such cases by turning the first failed attempt into actionable diagnostic memory, but it can also add irrelevant context, cost, and latency if the incident is simple or if the model lacks enough capability to use the feedback. Consequently, the central question is not whether reflection is conceptually useful, but under which operational conditions it improves verified remediation beyond a simple retry [2], [4].

This work studies Reflexion for Kubernetes incident remediation using a controlled experimental setup. We compare three configurations: a one-attempt baseline, a two-try setting without reflection, and a full Reflexion setting. Separating these conditions is necessary because simply allowing another attempt may improve results even when no explicit reflection is used. Thus, we distinguish the benefit of retrying from the benefit of reflection and test whether self-reflection contributes additional measurable value. Because the trial budget is two attempts, each run produces at most one reflection; the study therefore evaluates single-round, Reflexion-style reflection—the marginal value of the first reflection—rather than the full multi-trial Reflexion loop with accumulated episodic memory [4].

The study is guided by four research questions:

- **RQ1:** How much does a single round of Reflexion-style reflection improve success on Kubernetes troubleshooting tasks beyond a verified retry?
- **RQ2:** How does the effect differ between single-fault and multi-fault incidents?
- **RQ3:** How does model capability affect the value of reflection?
- **RQ4:** Which model–configuration pair provides the best balance of success, cost, and runtime?

The results indicate that Reflexion is best viewed as a conditional mechanism. Most of the overall improvement comes from allowing a second attempt, whereas reflection is most helpful for complex incidents with multiple interacting faults. This has practical implications for AIOps and DevOps systems: reflection can be saved for complex or unresolved cases instead of being applied automatically to every failure.

This paper makes the following contributions:

- We introduce an autonomous Kubernetes remediation architecture based on Reflexion, combining an LLM actor, deterministic evaluator, self-reflection module, and bounded episodic memory.
- We conduct 1,500 controlled runs over five Claude models, ten Kubernetes incidents, and three configurations.
- We disentangle retrying from reflection and show that the second attempt accounts for most aggregate improvement; a post-hoc self-verification ablation (Section IV-G) further shows that allowing the actor to verify its own fix recovers about half of this retry effect on single-fault scenarios, while an externally verified second attempt remains superior.
- We observe preliminary evidence that Reflexion may help multi-fault incidents and mid-capability models, while noting in Section V-E that the multi-fault finding requires confirmation on a larger scenario set. A post-hoc feedback-only control (Section IV-F) further shows that supplying the evaluator’s verbatim failure reason without any reflection is substantially more effective than reflection on identically instantiated multi-fault scenarios, attributing most of the benefit to the failure signal itself rather than to verbal self-reflection.
- We discuss cost, latency, and model-selection implications for production-focused AIOps deployments.

The remainder of this paper is organized as follows. Section II reviews related work on Kubernetes operations, LLM agents, self-reflection methods, and operational benchmarks. Section III describes the experimental design, agent architecture, prompts, scenarios, and evaluation procedure. Section IV reports the quantitative and qualitative results. Section V discusses their implications and the threats to validity, and Section VI concludes the paper and outlines future work.

II. RELATED WORK

This section provides a narrative review that positions the study within four strands of prior work: Kubernetes operations and AIOps, LLM agents and self-reflection methods,

benchmarks for operational agents, and recent journal studies on LLM-based operations. The review is selective rather than systematic; it covers the work most directly related to the research questions instead of following a formal search protocol.

A. KUBERNETES OPERATIONS AND AIOPS

Kubernetes grew out of earlier cluster-management systems and is now widely used for cloud-native deployment [8], [9]. Its declarative API and controller-driven design support scalable operation, but they also expand the space of possible configuration errors. Frequent failure patterns include image-pull failures, CrashLoopBackOff behavior, service endpoint mismatches, liveness-probe faults, volume and ConfigMap problems, and resource-scheduling issues.

AIOps seeks to automate fault detection, root-cause analysis, and remediation by using telemetry, logs, traces, and learned models [10]. Automated log analysis is a key part of reliability engineering for distributed systems [11]. Many current tools assist with monitoring or workflow automation, but they remain limited when unfamiliar combinations of misconfigurations occur. Within Kubernetes, K8sGPT provides AI-supported diagnostics [12], and newer multi-agent systems such as KubeIntellect investigate end-to-end management [13]. The specific contribution of self-reflection and episodic memory to autonomous Kubernetes remediation, however, has not been studied in depth.

B. LLM AGENTS AND SELF-REFLECTION

LLM agents augment language models with tools, memory, and repeated interaction with external environments [5], [14]. Multi-agent frameworks such as AutoGen extend this paradigm by orchestrating several conversing agents [15]; the present study instead evaluates a single-agent design, so that the effect of reflection is not confounded with inter-agent coordination. ReAct combines reasoning with action, enabling a model to choose what information to gather and what operation to perform [6]. Prior tool-learning research also shows that foundation models can use APIs and command-line interfaces when suitable constraints are provided [16].

Self-reflection methods try to improve future actions by examining previous failures. Self-Refine has a model critique and revise its own output [2]; Tree of Thoughts explores multiple reasoning paths [3]; and Reflexion records verbal feedback after unsuccessful trajectories [4]. Such methods are appealing for operations because they avoid retraining. Their success depends on whether the reflection is specific enough for the model to use on the next attempt.

An important design dimension of these methods is the number of reflection rounds. In the original Reflexion loop, the agent may iterate over many trials, accumulating several verbal reflections in a bounded episodic memory so that later attempts are conditioned on a growing body of self-generated guidance [4]. A single-round variant, in contrast, generates and consumes exactly one reflection between two attempts. The present study deliberately evaluates the single-

round setting: it isolates the marginal value of the first reflection, which is the most cost-relevant case for production remediation because every additional trial multiplies latency and API cost. Whether accumulated reflective memory yields compounding benefits over larger trial budgets is a separate question that this study does not measure.

C. BENCHMARKS FOR OPERATIONAL AGENTS

Broad software-engineering benchmarks, including HumanEval and SWE-bench, assess code generation or issue resolution [17], [18], and AgentBench evaluates LLMs as agents across interactive environments such as operating systems and databases [19]; cloud operations, however, require interaction with a specific environment and external validation. Kubernetes-focused benchmarks such as KubeLLM-Bench cover common faults such as port mismatches, selector mistakes, liveness-probe errors, incorrect image names, and missing environment variables [20]. AIOpsLab likewise evaluates autonomous cloud agents with realistic fault injection [21], and ITBench benchmarks AI agents on real-world IT automation tasks, including site-reliability scenarios on Kubernetes, and reports that even the strongest agents resolve only a small fraction of them [22]. This study uses Kubernetes misconfiguration scenarios and adds multi-fault cases to test whether reflection helps when faults must be diagnosed in sequence.

D. RECENT AIOPS AND LLM-BASED OPERATIONS STUDIES

Diaz-de-Arcaya *et al.* [23] provide a systematic journal survey of MLOps and AIOps and frame both areas as complementary operational disciplines for deploying, monitoring, and maintaining intelligent systems. Their review emphasizes that organizational adoption requires more than model accuracy: collaboration, lifecycle governance, automation pipelines, and operational feedback loops are also necessary. Autonomous Kubernetes remediation fits this view: detection, diagnosis, repair, and post-failure learning form one repeatable operational process instead of isolated model calls.

Zhang *et al.* [24] survey AIOps in the era of large language models and organize recent work around failure data sources, AIOps tasks, LLM-based methods, and evaluation practices. Their analysis shows that LLMs are now used for incident understanding, runbook assistance, and operator-facing decision support as well as for text interpretation. Kubernetes incident remediation is such a setting: the model must combine heterogeneous evidence, choose actions, and remain verifiable under cost, latency, and safety constraints.

Zha *et al.* [25] study LLM-assisted alert aggregation for AIOps and propose a two-phase approach that first groups alerts by temporal and spatial locality and then uses an LLM to trace cascading service-failure effects. Their results show that LLMs can add value when operational evidence must be connected across dependent services, especially during alert storms. Multi-fault Kubernetes remediation poses the same

problem: the system must look beyond a single symptom and decide whether several observations share one cause.

Zhou *et al.* [26] examine the use of large language models and BERT for log parsing and anomaly detection. Their journal article treats logs as a primary operational signal and shows that combining LLM-based parsing with transformer-based anomaly detection can improve downstream fault-analysis performance. Remediation agents likewise depend on logs and event streams to infer why pods crash, services lose endpoints, or configurations fail, so the quality of log interpretation directly affects repair success.

Ma *et al.* [27] introduce AdaptiveLog, a journal framework that coordinates large and small language models for adaptive log analysis. The key idea is that different log-analysis subtasks do not always require the same model capacity, so a system can combine cheaper lightweight models with more capable models when deeper reasoning is needed. Our model-selection results point the same way: mid-tier models offer a strong cost-performance trade-off, and high-capability models should be reserved for cases where their additional cost is justified.

LogLAA [28] proposes an adaptive integrated log anomaly analysis framework that combines log parsing, anomaly detection, and LLM-based anomaly interpretation. The study raises a practical issue for operational LLM use: feeding large volumes of raw logs directly into a model is costly and can reduce analysis quality, whereas structured filtering and targeted anomaly extraction can make LLM explanations more useful. The verification-driven design used here follows the same principle: the agent is guided by structured evidence and deterministic evaluation instead of free-form reasoning alone.

Table 1 summarizes the LLM-based Kubernetes-oriented systems most closely related to this study and contrasts them along the dimensions that define our design: reasoning strategy, memory use, tool orchestration, and verification. The comparison shows the gap this study addresses: prior systems diagnose without applying repairs, coordinate several agents without isolating the contribution of self-reflection, or benchmark agents without a learning-from-failure mechanism. None of them separates the value of a verified retry from the value of verbal self-reflection, which is the central question of this paper.

III. MATERIALS AND METHODS

This section describes the controlled study design, the four-component agent architecture, the prompt design, the ten fault scenarios, the evaluated models and configurations, and the metrics and statistical procedures used in the analysis.

A. STUDY DESIGN

We use a controlled design where the independent variables are the agent configuration, the model/capability tier, and the incident scenario. The main dependent variable is binary remediation success. Additional measurements include the number of trials consumed, execution time, token use, and

API cost. To account for variability in LLM behavior, every model–configuration–scenario combination was run ten times.

The experiment compares three configurations. In the baseline, the agent receives one attempt and no reflection. In the two-try setting, the agent receives a second attempt but no self-reflection, allowing the retry effect to be measured separately. In the full Reflexion setting, the agent receives a second attempt together with natural-language feedback produced after the failed first attempt. Because the budget is fixed at two attempts, each run yields at most one reflection; this isolates a single reflection between two tries rather than the cumulative effect of Reflexion’s multi-trial memory loop. With ten scenarios, five models, three configurations, and ten repetitions, the experiment comprises 1,500 runs. The complete thesis version of the experimental protocol, implementation details, appendices, and raw-data schema is documented in the corresponding master’s thesis [29].

The full Reflexion condition combines two factors: access to an external failure signal and the LLM-generated reflection itself. To separate them, we additionally ran a post-hoc *feedback-only* control configuration. In this control, the second attempt is preceded by no reflection call; instead, the episodic memory receives a single verbatim string of the form “The previous attempt failed. Evaluator feedback: {reason} (Pod status: {status})”, where *reason* is the programmatic evaluator’s textual failure explanation. The Actor prompt and memory-injection mechanism are identical across all retry-capable configurations; only the memory content differs. Because Claude 3 Haiku and Claude 3.5 Haiku had been retired from the Anthropic API by the time of revision, this control was run on the three models that remained available (Haiku 4.5, Sonnet 4.5, and Opus 4.5) on the multi-fault scenarios, where the reflection benefit was concentrated (Section IV-F).

A second post-hoc ablation addresses the complementary confound on the baseline side. Because the Actor is prohibited from verifying its own fix (Section III-C), the baseline measures a single *unverified* attempt, so the difference between the baseline and the two-try condition bundles an additional attempt with an external verification signal. The *self-verification* configuration keeps the single-attempt, no-reflection design of the baseline but replaces the prompt’s stop instruction with an explicit allowance to verify the fix and, if the problem persists, to repeat the diagnose–fix–verify cycle at most three times within the same context window. This ablation was run on the same three models over the eight single-fault scenarios with five repetitions per cell (120 runs) and is compared against the matrix baseline and two-try runs in the same cells (Section IV-G).

B. AGENT ARCHITECTURE

As illustrated in Fig. 1, the remediation system has four components. The Actor is an LLM-driven Kubernetes troubleshooting agent built with the Claude Agent SDK, version

TABLE 1. Related LLM-based systems for Kubernetes and cloud operations.

System	Reasoning strategy	Memory	Tool orchestration	Verification	Gap addressed here
K8sGPT [12]	Rule-based analyzers with LLM-generated explanations	None	Built-in cluster scanners	None; operator applies fixes	Diagnosis only; no autonomous repair loop
KubeIntellect [13]	Supervisor-coordinated multi-agent workflow	Session context	LLM-orchestrated <code>kubectl</code> and API tools	Human oversight in the loop	End-to-end management; reflection not isolated
KubeLLMBench [20]	Multi-agent with RAG over Kubernetes documentation	Chat history	<code>kubectl</code> tool calls	Benchmark checks	No learning-from-failure mechanism under test
AIOpsLab [21]	Agent-agnostic evaluation harness	Depends on agent	Fault-injection framework APIs	Programmatic oracles	Benchmark infrastructure, not an agent design
This work	Single ReAct-style actor with Reflexion	Bounded episodic memory of verbal reflections	Claude Agent SDK with <code>kubectl</code>	Deterministic programmatic evaluator	Disentangles verified retry from self-reflection

TABLE 2. Programmatic evaluator checks.

Check	Evidence inspected	Failure signal
Pod phase	Pod status from Kubernetes API	Not running
Readiness	Ready container count	Not all ready
Error state	Events and crash/image errors	Error reason present
Reachability	Endpoints or HTTP test	No endpoint or failed request

0.1.6 [30]. It investigates incidents through `kubectl`, applies repairs, and ends with a fixed completion signal. The Actor prompt follows three phases: diagnosis with `kubectl get`, `describe`, `logs`, and `get events`; repair with non-interactive commands such as `patch`, `set image`, `apply`, and `delete`; and termination with a fixed `FIX COMPLETE` signal. The prompt explicitly forbids interactive editors such as `kubectl edit`, `vim`, `nano`, and `vi`, and it instructs the Actor not to enter retry loops because external validation is handled separately.

The Evaluator is deterministic and programmatic, checking cluster state without any LLM call. The Actor never receives the Evaluator’s binary success/failure signal; it observes only raw cluster state, including pod status, events, logs, and endpoints, so success labels cannot leak into the Actor’s remediation policy. The Evaluator verifies that pods are running, all containers are ready, relevant error states are absent, and service endpoint or HTTP connectivity requirements are met. This avoids both evaluation nondeterminism and extra evaluation cost. Table 2 summarizes the checks used to decide whether a run is successful or should continue to another attempt.

If an attempt fails under the full Reflexion configuration, the Self-Reflection component makes one LLM call to analyze the Actor’s trajectory and the evaluator result. The reflection prompt includes the attempted trajectory, failure reason, pod status, readiness information, and previous episodic memories. It requires three structured parts: credit

assignment, root-cause analysis, and concrete next steps. Reflections are written in the first person and are constrained to be specific, actionable, analytical, and concise. This text is saved in episodic memory and passed to the Actor on the next trial. Episodic memory is reset at the start of every run, so no reflection is shared across repetitions, scenarios, or models. Because the present protocol permits at most two attempts per run, only one reflection can be generated within a run; the three-reflection cap is an implementation safeguard inherited from the general Reflexion framework and is not exercised by the reported two-attempt experiments.

C. PROMPT DESIGN

The Actor follows a three-phase system prompt: (1) *diagnose* with read-only `kubectl get`, `describe`, `logs`, and `get events`; (2) *fix* with non-interactive `patch`, `set image`, `apply`, or `delete`; and (3) *stop* by emitting `FIX COMPLETE`. The prompt explicitly forbids interactive editors (`kubectl edit`, `vim`, `nano`, `vi`) and instructs the Actor not to verify its own fix or enter retry loops, because verification is delegated to the deterministic Evaluator. The self-verification ablation (Section IV-G) uses a variant of this prompt in which Phase 3 instead instructs the Actor to re-check the pod after the fix and, if the problem persists, to return to diagnosis and repeat the diagnose–fix–verify cycle at most three times before emitting `FIX COMPLETE`; all other prompt content is identical.

The Self-Reflection module is prompted to act as a verbal reinforcement signal. Its system prompt requires the reflection to be specific, actionable, analytical, first-person, and concise (200–400 words). Its user template is populated with the failed trajectory, the Evaluator reason, pod status and readiness, and prior reflections, and it mandates three structured parts: (i) *credit assignment* (which action or decision caused the failure), (ii) *root cause* (what state was misunderstood or missed), and (iii) *concrete next steps* (what to check or change next). These reflections are stored in episodic memory and retrieved ($\text{top-}k = 3$) for the following trial.

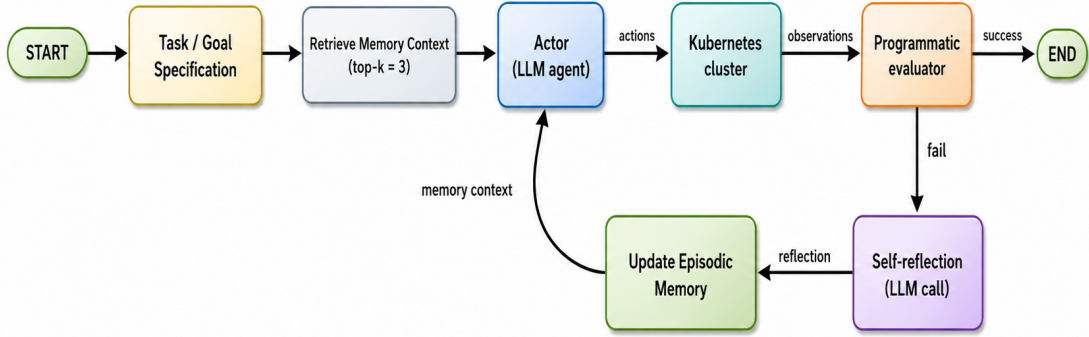


FIGURE 1. Reflexion-based remediation architecture.

D. TEST SCENARIOS

The benchmark includes ten Kubernetes misconfiguration scenarios. Eight are single-fault cases derived from Kubernetes troubleshooting benchmarks, and two are multi-fault cases intended to resemble production incidents in which an initial repair exposes or leaves other faults. Case 9 combines an image tag typo with a service `targetPort` mismatch: after the image tag is corrected, the service can still route traffic to the wrong port. Case 10 combines an image typo, a missing ConfigMap mount, and a wrong service `targetPort`, requiring several resources to be repaired in sequence. Automated setup and cleanup scripts ensure that every run begins from an identical state. Table 3 summarizes these cases and makes the 8/1/1 split across single-, double-, and triple-fault scenarios explicit. Instead of a priori difficulty labels, the table reports the empirical success rate observed across all models and configurations, because hand-assigned labels aligned only loosely with the difficulty actually observed in the experiments.

E. MODELS AND CONFIGURATIONS

We evaluated five Anthropic Claude models representing different capability levels, listed here with their official names: Claude 3 Haiku, Claude 3.5 Haiku, Claude Haiku 4.5, Claude Sonnet 4.5, and Claude Opus 4.5 (the exact API identifiers are given in the Data and Code Availability section). For brevity, tables and figures refer to them as Haiku 3.0, Haiku 3.5, Haiku 4.5, Sonnet 4.5, and Opus 4.5. These models are treated as a spectrum from lightweight to high-capability systems. The evaluation was deliberately restricted to a single model family: the Actor’s tool-use behavior is mediated by the Claude Agent SDK, so holding the SDK, prompts, and tool

interface constant isolates the effect of model capability from framework differences, at the acknowledged cost of cross-family generality (Section V-C). All runs used the Claude Agent SDK’s default API parameters. Table 4 summarizes the three main experimental configurations and the two post-hoc controls (feedback-only and self-verification) described in Section III-A.

F. EVALUATION METRICS AND STATISTICAL ANALYSIS

The main metric is success rate, defined as the share of runs that the programmatic evaluator judged to be remediated. For configuration c , success rate is calculated as

$$SR_c = \frac{1}{N_c} \sum_{i=1}^{N_c} \mathbb{I}(S_i = \text{success}), \quad (1)$$

where N_c denotes the number of runs for configuration c , S_i is the outcome for run i , and $\mathbb{I}(\cdot)$ denotes the indicator function.

The second-attempt effect is measured as the difference between the two-try and baseline settings. Reflexion benefit is measured as the difference between full Reflexion and the two-try setting without reflection. Cost is computed from token counts and model prices, and execution time is recorded in seconds. The experiment logs also store trials used, total token counts, command trajectories, evaluator reasons, and per-run JSON records that were exported into an SQLite database for analysis. Statistical comparisons use chi-square independence tests, Wilson 95% confidence intervals for proportions, and Cohen’s h to interpret effect sizes. These tests treat runs as Bernoulli observations; however, repeated runs of the same model–scenario pair and differences in scenario difficulty introduce clustering, so the pooled inferential results should be interpreted as descriptive. To address this, Section IV-

TABLE 3. Evaluated Kubernetes incidents.

Case	Scenario	Faults	Main symptom	Overall SR
1	Wrong port	1	Pod running but unreachable through the service	38.7%
2	Incorrect selector	1	Service has no endpoints	67.3%
3	Liveness probe	1	Pod enters CrashLoopBackOff after probe failures	64.0%
4	Wrong interface	1	Application binds to localhost and cannot be reached by service	60.7%
5	Port mismatch	1	Connection refused due to inconsistent ports	74.0%
6	Misspelling	1	Typo in image, label, or selector causes a failure symptom	92.0%
7	Volume mount	1	CreateContainerConfigError due to ConfigMap reference mismatch	44.0%
8	Environment variable	1	Application crashes at startup due to missing variables	70.7%
9	Double trouble	2	Image tag typo plus service targetPort mismatch	45.3%
10	Triple threat	3	Image typo, missing ConfigMap mount, and wrong targetPort	30.7%

Overall SR: empirical success rate over all models and configurations (150 runs per case; see Table 7).

TABLE 4. Experimental configurations.

Config.	Trials	Refl.	Purpose
Baseline	1	No	Measure one-shot remediation performance
Two-try	2	No	Measure retry value separately from reflection
Full Reflexion	2	Yes	Measure reflection and memory value
Feedback-only	2	No	Isolate evaluator feedback from reflection (post-hoc control)
Self-verify	1	No	Permit in-attempt verification and revision (post-hoc ablation)

G additionally reports a mixed-effects logistic model with random intercepts for scenario and model and a clustered generalized estimating equation fitted on the per-run records.

IV. EXPERIMENTS AND RESULTS

This section reports the empirical findings: overall configuration performance, model- and scenario-level breakdowns, the retry-versus-reflection ablation, the complexity analysis, cost and runtime measurements, statistical tests, and a qualitative analysis of the generated reflections.

A. OVERALL PERFORMANCE

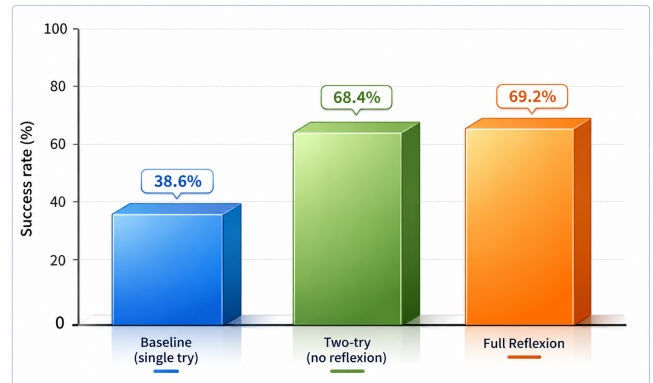
Table 5 and Fig. 2 present the aggregate results. The one-attempt baseline achieved a 38.6% success rate. Providing a second attempt without reflection, triggered by the external evaluator's failure signal, raised success to 68.4%, an improvement of 29.8 percentage points. Full Reflexion produced the highest overall success rate at 69.2%, but this was only 0.8 percentage points above the non-reflective two-try setting. Because Reflexion adds another LLM call after failed attempts, it also increased both mean runtime and cost.

B. MODEL-LEVEL PERFORMANCE

Remediation success was strongly influenced by model capability, as shown in Table 6. Opus 4.5 achieved the best overall

TABLE 5. Aggregate configuration performance.

Configuration	Runs	Succ.	SR	Cost	Time
Baseline	500	193	38.6%	\$0.108	83 s
Two-try	500	342	68.4%	\$0.155	110 s
Full Reflexion	500	346	69.2%	\$0.179	173 s

**FIGURE 2.** Overall success by configuration.

success rate, whereas Sonnet 4.5 and Haiku 4.5 performed at similar levels. The older and less capable Haiku models lagged well behind. The cost-efficiency results show that the strongest model is not automatically the most economical option: among the models with strong performance, Haiku 4.5 combined high success with the lowest average cost.

C. SCENARIO-LEVEL PERFORMANCE

Incident difficulty differed considerably across scenarios, as reported in Table 7. Typographical errors and selector or port cases were generally easier, whereas wrong-port, volume-mount, and multi-fault cases were more challenging. The triple-fault case had the lowest overall success rate, suggesting that concurrent defects substantially weaken autonomous remediation reliability.

D. ABLATION: RETRY VERSUS REFLEXION

Table 9 separates the contribution of retrying from the contribution of explicit reflection. Every model benefited from

TABLE 6. Model-level success and cost.

Model	Success rate	Avg. cost	Runs
Claude Opus 4.5	87.3%	\$0.191	300
Claude Sonnet 4.5	73.3%	\$0.184	300
Claude Haiku 4.5	73.0%	\$0.105	300
Claude Haiku 3.5	34.7%	\$0.080	300
Claude Haiku 3.0	25.3%	\$0.176	300

TABLE 7. Test-case success rates.

Case	Description	Faults	SR
case6	Image typo	1	92.0%
case5	Port mismatch	1	74.0%
case8	Environment variable	1	70.7%
case2	Incorrect selector	1	67.3%
case3	Liveness probe	1	64.0%
case4	Wrong interface	1	60.7%
case9	Double trouble	2	45.3%
case7	Volume mount	1	44.0%
case1	Wrong port	1	38.7%
case10	Triple threat	3	30.7%

a second attempt, with improvements ranging from +15 to +37 percentage points. Reflexion produced clear additional gains only for mid-capability models: Haiku 4.5 increased from 84% to 88%, and Sonnet 4.5 increased from 84% to 87%. Opus 4.5 did not improve further because it had already reached 98% with two attempts. Haiku 3.0 and Haiku 3.5 showed negligible or slightly negative changes, indicating that weaker models may create reflections they cannot use effectively. Fig. 3 visualizes this non-linear relationship between model capability and reflection benefit.

Table 8 reports the attempt-level breakdown underlying these aggregates, restoring the recovery-rate analysis from the thesis protocol [29]. The table supports three observations. First, first-attempt success in the retry-capable configurations (38.0% and 40.2%) is consistent with the one-attempt baseline (38.6%), confirming that the conditions differ only in what happens after a failed first attempt. Second, the full Reflexion condition generated exactly 310 reflections, one per failed first attempt; no reflection follows a failed second attempt because no further trial exists. Third, the recovery rate after a failed first attempt was 50.3% (156/310) with reflection versus 47.2% (141/299) without, a modest difference consistent with the small aggregate gap between the two configurations.

E. PROBLEM COMPLEXITY AND REFLEXION BENEFIT

Grouping scenarios by complexity gives the clearest, though preliminary, signal for Reflexion. As shown in Table 10 and Fig. 4, Reflexion lowered success by 2.0 percentage points for single-fault cases, but raised success by 12.0 percentage points for the double-fault case and by 12.0 percentage points for the triple-fault case. Because each multi-fault category contains only one scenario, these values should be interpreted as scenario-level evidence that motivates further testing, not as a definitive complexity effect. The pattern suggests that

TABLE 8. Attempt-level breakdown of the retry-capable configurations.

Model	1st	Refl.	2nd	Fail	SR
<i>Full Reflexion</i>					
Haiku 3.0	22	78	7	71	29%
Haiku 3.5	16	84	28	56	44%
Haiku 4.5	41	59	47	12	88%
Sonnet 4.5	44	56	43	13	87%
Opus 4.5	67	33	31	2	98%
Total	190	310	156	154	69.2%
<i>Two-try (no reflection)</i>					
Haiku 3.0	16	–	15	69	31%
Haiku 3.5	17	–	28	55	45%
Haiku 4.5	49	–	35	16	84%
Sonnet 4.5	50	–	34	16	84%
Opus 4.5	69	–	29	2	98%
Total	201	–	141	158	68.4%

1st = successful first attempts; Refl. = reflections generated; 2nd = successful second attempts; Fail = final failures. Each model-configuration cell comprises 100 runs.

TABLE 9. Model-level ablation results (Δ Two = Two-try minus Baseline success rate; Δ Refl. = Full Reflexion minus Baseline success rate, in percentage points).

Model	Base	Two	Refl.	Δ Two	Δ Refl.
Opus 4.5	66%	98%	98%	+32 pp	+32 pp
Sonnet 4.5	49%	84%	87%	+35 pp	+38 pp
Haiku 4.5	47%	84%	88%	+37 pp	+41 pp
Haiku 3.5	15%	45%	44%	+30 pp	+29 pp
Haiku 3.0	16%	31%	29%	+15 pp	+13 pp

reflection may be most beneficial when an incomplete repair forces the agent to revise its causal explanation or when several resources must be fixed sequentially.

F. FEEDBACK-ONLY CONTROL ON MULTI-FAULT SCENARIOS

The full Reflexion condition combines two factors: an external failure signal and the LLM-generated verbal reflection. To separate them, the post-hoc feedback-only control (Section III-A) supplies the second attempt with the evaluator’s verbatim failure reason and pod status, but no reflection. The control comprises 142 runs on the three current models: the two multi-fault scenarios (110 runs) and an additional exploratory two-fault scenario outside the ten-scenario matrix (32 runs), discussed below.

One difference in scenario provenance constrains the comparison. The control ran on the current replication-package instantiation of the multi-fault scenarios, which differs from the instantiation used for the original 1,500-run matrix in one respect: the agent’s diagnostic entry point. In the matrix runs, the agent was pointed at a neighboring client pod and had to explore the namespace to locate the faults; in the current instantiation, the agent is pointed directly at the faulty deployment. This change substantially increases difficulty, because agents entering at the faulty pod typically repair the pod-level fault, observe a Running pod, and terminate without examining the service path. All comparisons in this

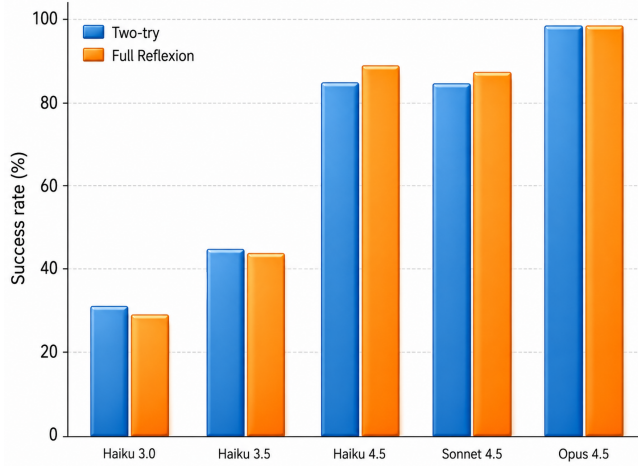


FIGURE 3. Model capability and Reflexion benefit.

TABLE 10. Reflexion benefit by problem complexity.

Complexity	Cases	Two-try	Reflexion	Benefit
Single-fault	case1–8	72.8%	70.8%	-2.0 pp
Double-fault	case9	62.0%	74.0%	+12.0 pp
Triple-fault	case10	40.0%	52.0%	+12.0 pp

subsection therefore use only runs executed on the identical current instantiation (the post-freeze replication runs of the two-try and full Reflexion arms and the new feedback-only runs), and the values in Table 11 are consequently not comparable to the matrix values in Table 10.

Table 11 reports the outcome. Under this instantiation, the plain two-try condition never recovered on the multi-fault scenarios (0/50), full Reflexion recovered a minority of runs (13/50, 26.0%), and the feedback-only control recovered 84/110 (76.4%). The feedback-only advantage is significant against both plain two-try ($\chi^2(1) = 80.4, p < 0.001$) and full Reflexion ($\chi^2(1) = 36.5, p < 0.001$), and it replicates on the exploratory two-fault scenario (a ConfigMap reference combined with a selector mismatch): 96.9% (31/32) versus 54.8% (17/31) for Reflexion ($\chi^2(1) = 15.3, p < 0.001$). Reflexion itself still improved markedly over plain retry under this instantiation ($\chi^2(1) = 14.9, p < 0.001$), confirming that reflection transmits part of the failure signal; the verbatim feedback simply transmits it more completely. The pattern is capability-dependent in the same direction as the main results: on the two multi-fault scenarios, Sonnet 4.5 (40/40) and Opus 4.5 (30/30) converted the verbatim failure reason into a successful repair in every run, whereas Haiku 4.5 remained well below ceiling on Case 9 (10/25) and Case 10 (4/15), indicating that an explicit failure signal does not compensate for limited tool-use capability.

A second caveat is that the control receives more information than the reflection module. In the implemented pipeline, the reflection module conditions on the failed trajectory and the pod status but not on the evaluator's textual failure reason,

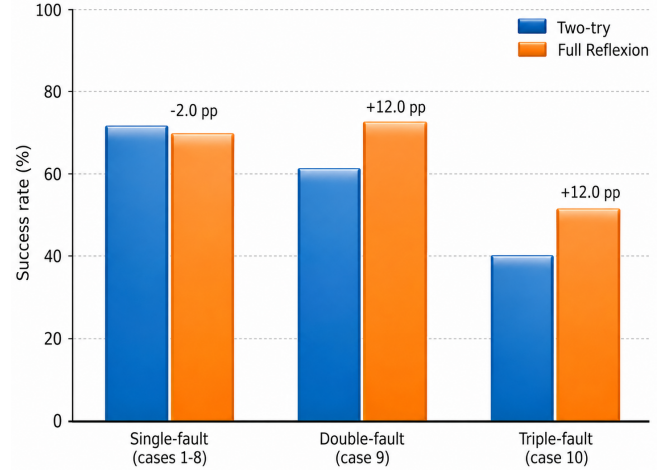


FIGURE 4. Complexity-level Reflexion gain.

TABLE 11. Feedback-only control on the current scenario instantiation.

Config.	Case 9	Case 10	Case 11
Two-try	0.0% (0/35)	0.0% (0/15)	53.1% (17/32)
Full Reflexion	34.3% (12/35)	6.7% (1/15)	54.8% (17/31)
Feedback-only	76.9% (50/65)	75.6% (34/45)	96.9% (31/32)

Models: Haiku 4.5, Sonnet 4.5, and Opus 4.5. Cases 9–10 are the multi-fault scenarios in their current instantiation (see text); Case 11 is an exploratory two-fault scenario outside the ten-scenario matrix. All three configurations are evaluated on the identical instantiation, so these values are not comparable to Table 10.

whereas the feedback-only memory contains that reason verbatim. The comparison therefore does not show that verbal reflection is harmful; it shows that the evaluator's signal is what matters most and that the reflection stage, as implemented here, does not reliably reconstruct it from the trajectory alone.

G. SELF-VERIFICATION ABLATION ON SINGLE-FAULT SCENARIOS

The baseline prohibits the Actor from verifying its own fix, so the retry effect reported in Section IV-D combines two factors: a second attempt and an external verification signal. To bound the share attributable to verification alone, the post-hoc self-verification ablation (Section III-A) gives the Actor a single attempt without reflection but permits up to three diagnose–fix–verify cycles within that attempt. The ablation comprises 120 runs (the three current models on the eight single-fault scenarios, five repetitions per cell) and is compared against the matrix baseline and two-try runs in the same 24 model–scenario cells (240 runs each). The single-fault scenarios were not modified between the matrix runs and the revision, so, unlike the feedback-only control, this comparison uses the identical scenario instantiation.

Table 12 reports the outcome. Permitting self-verification raised single-attempt success from 67.5% (162/240) to 80.8% (97/120; $\chi^2(1) = 7.05, p = 0.008$), recovering roughly half of the 25.0-percentage-point gap between the baseline and the

two-try condition in these cells (92.5%, 222/240). The externally verified second attempt nevertheless remained significantly better than the self-verifying single attempt ($\chi^2(1) = 10.8$, $p = 0.001$). The effect is capability-dependent in the same direction as the main results: Haiku 4.5 did not benefit (58.8% versus 60.0%, $p = 0.90$), whereas Sonnet 4.5 (61.2% to 82.5%, $\chi^2(1) = 5.56$, $p = 0.018$) and Opus 4.5 (82.5% to 100%, Fisher's exact $p = 0.005$) converted self-checks into successful repairs; for Sonnet 4.5 the self-verifying single attempt was statistically indistinguishable from its two-try result (87.5%, $p = 0.46$).

Two scenario-level details qualify this result. First, the gain is concentrated in Case 7 (volume mount), where the baseline succeeded in 4 of 30 runs but self-verification in 14 of 15. In all 30 baseline runs the agents applied the correct repair (creating the missing ConfigMap) and terminated after a median of 40 seconds, while the pod was still `Pending` waiting for the volume to mount; because the Evaluator samples the cluster immediately after the Actor stops, 26 of these runs were scored as failures. Self-verifying agents instead polled the pod until it reached `Running` (median 83 seconds). Part of the measured value of verification is therefore the value of waiting for the cluster to converge rather than of diagnostic correction. Excluding Case 7, self-verification changed success from 75.2% to 79.0% ($\chi^2(1) = 0.57$, $p = 0.45$), whereas the two-try condition reached 94.3% and remained superior ($\chi^2(1) = 16.9$, $p < 0.001$). Second, self-verification can mislead. In Case 4 (wrong interface), Haiku 4.5 failed in all five self-verifying runs but succeeded in 9 of 10 baseline runs: the self-verifying agents probed the application with `curl` against `localhost` from inside the pod, which succeeded because the application was bound to the loopback interface, and accepted this local check as confirmation although the service remained unreachable from outside the pod, which is exactly what the external Evaluator tests. Self-verification is also more expensive: \$0.28 and 139 seconds per run, against \$0.12 and 94 seconds for the baseline and \$0.16 and 100 seconds for two-try, because every verification cycle extends a single, growing context window. The ablation therefore indicates that the retry effect should be read as the value of an additional, externally verified attempt, of which in-attempt self-verification captures only part, chiefly for stronger models and at higher cost.

H. COST AND EXECUTION TIME

Table 13 reports cost efficiency for the retry-capable configurations only. Haiku 4.5 had the lowest cost per successful remediation while still maintaining a high success rate. Opus 4.5 achieved the highest absolute success, but with a higher average cost. Although Haiku 3.0 is lightweight, it was inefficient because its low success rate increased the cost of each successful remediation.

Runtime measurements reveal the same trade-off. Since Reflexion makes an extra LLM call after failures, it increases average execution time. For instance, mean runtime rose from 110 seconds in the two-try setting to 173 seconds in the full

TABLE 12. Self-verification ablation on the single-fault scenarios.

Model	Baseline	Self-verify	Two-try
Haiku 4.5	58.8% (47/80)	60.0% (24/40)	90.0% (72/80)
Sonnet 4.5	61.2% (49/80)	82.5% (33/40)	87.5% (70/80)
Opus 4.5	82.5% (66/80)	100.0% (40/40)	100.0% (80/80)
All	67.5% (162/240)	80.8% (97/120)	92.5% (222/240)
Excl. Case 7	75.2% (158/210)	79.0% (83/105)	94.3% (198/210)
Cost per run	\$0.12	\$0.28	\$0.16

Models: Haiku 4.5, Sonnet 4.5, and Opus 4.5 on Cases 1–8. Baseline and Two-try are the matrix runs in the same model-scenario cells (ten repetitions per cell); Self-verify is the post-hoc single-attempt configuration with in-attempt verification permitted (five repetitions per cell). All three arms use the identical scenario instantiation.

TABLE 13. Cost per successful remediation.

Model	Success rate	Avg. cost	Cost/success
Haiku 4.5	86.0%	\$0.116	\$0.135
Haiku 3.5	44.5%	\$0.091	\$0.205
Opus 4.5	98.0%	\$0.209	\$0.213
Sonnet 4.5	85.5%	\$0.204	\$0.239
Haiku 3.0	30.0%	\$0.214	\$0.713

Reflexion setting. This latency cost supports using reflection selectively for complex, unresolved, or high-impact incidents instead of simple single-fault problems.

I. STATISTICAL ANALYSIS

The overall gain from baseline to two-try was statistically significant, and the gain from baseline to full Reflexion was also significant. In contrast, the aggregate difference between two-try and full Reflexion was not statistically significant. Table 15 gives the chi-square comparisons and effect sizes, while Table 17 shows that the confidence intervals for two-try and full Reflexion overlap substantially. The complexity-specific Reflexion improvement in the multi-fault cases should therefore be interpreted cautiously: the experimental analysis indicated that roughly 150–200 additional samples per condition would be needed to test that effect size with stronger statistical power at $\alpha = 0.05$.

To avoid the pseudo-replication induced by treating the 500 correlated runs per configuration as independent, we additionally analyzed the configuration effects using the model as the unit of analysis ($n = 5$), as summarized in Table 16. Paired comparisons across models confirmed the pooled findings: the retry effect was large and significant (mean +29.8 pp, paired $t(4) = 7.66$, $p = 0.002$, Cohen's $d_z = 3.43$), whereas the additional effect of reflection over a plain second attempt was small and not significant (mean +0.8 pp, $t(4) = 0.69$, $p = 0.53$, $d_z = 0.31$). Because this model-level test does not rely on the independence assumption, it strengthens the central conclusion that retrying rather than reflecting accounts for the aggregate improvement.

To complete this analysis, we fitted the per-run records with (a) a Bayesian mixed-effects logistic regression with random intercepts for model and scenario and (b) a logistic

TABLE 14. Mixed-effects and GEE configuration contrasts.

Contrast	Mixed OR [95% CI]	<i>p</i>	GEE OR [95% CI]	<i>p</i>
Two-try vs. Baseline	9.06 [6.19, 13.26]	< 0.001	3.44 [2.33, 5.08]	< 0.001
Reflexion vs. Baseline	9.55 [6.51, 14.00]	< 0.001	3.54 [2.27, 5.54]	< 0.001
Reflexion vs. Two-try	1.07 [0.75, 1.52]	0.71	1.03 [0.82, 1.30]	0.80

Mixed = Bayesian mixed-effects logistic regression on per-run outcomes (conditional odds ratios); GEE = logistic GEE clustered on model–scenario cells (marginal odds ratios); the two differ by construction but support identical conclusions.

TABLE 15. Pooled configuration significance tests.

Comparison	χ^2	<i>p</i>	Cohen's <i>h</i>	Sig.
Baseline vs. Two-try	89.24	< 0.001	0.607	Yes
Baseline vs. Reflexion	94.21	< 0.001	0.624	Yes
Two-try vs. Reflexion	0.07	0.796	0.017	No

TABLE 16. Model-as-unit paired tests.

Comparison	Mean Δ (pp)	<i>t</i> (4)	<i>p</i>	Cohen's <i>d_c</i>	Sig.
Baseline vs. Two-try	+29.8	7.66	0.002	3.43	Yes
Two-try vs. Full Reflexion	+0.8	0.69	0.53	0.31	No
Baseline vs. Full Reflexion	+30.6	6.26	0.003	2.80	Yes

generalized estimating equation (GEE) with an exchangeable working correlation clustered on model–scenario cells. Table 14 summarizes the configuration contrasts. Both estimators agree with the pooled and model-as-unit analyses: the retry effect is large and significant, whereas the additional effect of reflection over a plain second attempt is not. The estimated random-effect standard deviations for model (1.59) and scenario (1.40) are both substantial, confirming that the clustered analysis is warranted. Finally, a GEE interaction model restricted to the retry-capable configurations estimates the reflection-by-multi-fault interaction at $\beta = 0.558$ (SE 0.281, $p = 0.047$): the complexity-specific Reflexion benefit of Section IV-E remains statistically significant under clustering, although it still rests on only two multi-fault scenarios and therefore retains the hypothesis-generating caveat.

J. EXPLORATORY QUALITATIVE ANALYSIS OF REFLECTIONS

To examine the mechanism behind the quantitative results, we conducted an exploratory automated content analysis of the 310 reflection texts generated in the full-Reflexion condition of the experimental matrix (one reflection per failed first attempt; no reflection follows a failed second attempt). Of these, 156 preceded a successful second attempt (“recovered”) and 154 did not. We report three surface metrics per model: mean character length (Len.), the share of reflections containing at least one Markdown heading as a proxy for structured output (Head.), and the share containing an explicit credit-assignment section (Credit, detected by the case-insensitive marker *credit assignment*). Table 18 summarizes these characteristics, and Fig. 5 contrasts a representative recovered reflection with a representative unrecovered one. Because these metrics are automated surface measures rather than a validated human-coded scheme, we treat this analysis

TABLE 17. Success-rate confidence intervals.

Configuration	Success rate	95% CI
Baseline	38.6%	[34.3%, 42.9%]
Two-try	68.4%	[64.2%, 72.3%]
Full Reflexion	69.2%	[65.0%, 73.1%]

TABLE 18. Reflection characteristics by model.

Model	N	Recov.	Len.	Head.	Credit
Haiku 3.0	78	9.0%	1949	0.0%	20.5%
Haiku 3.5	84	33.3%	1906	14.3%	52.4%
Haiku 4.5	59	79.7%	2280	100.0%	100.0%
Sonnet 4.5	56	76.8%	2140	100.0%	82.1%
Opus 4.5	33	93.9%	2126	100.0%	100.0%

$N = 310$ in-matrix reflections. Recov. = share of reflections followed by a successful second attempt; Len. = mean length in characters; Head. = share containing at least one Markdown heading; Credit = share containing an explicit credit-assignment section.

as exploratory and hypothesis-generating rather than confirmatory.

The pattern supports the model-capability result: stronger and mid-tier models produced longer and more structured reflections, while lower-capability models more often produced vague failure acknowledgments that did not translate into effective follow-up actions.

A keyword comparison, normalized to occurrences per 1,000 words within each group and tested with χ^2 tests on the underlying counts, showed that recovered reflections were markedly more action-oriented. Terms associated with validation and concrete follow-up appeared significantly more often in recovered reflections: “verify” 64% more frequently (10.0 vs. 6.1 per 1,000 words, $p < 0.001$), “test” 98% more (9.8 vs. 5.0, $p < 0.001$), “wait” 475% more (3.3 vs. 0.6, $p < 0.001$), and “confirm” 80% more (4.9 vs. 2.7, $p < 0.001$). Conversely, the diagnostic label “incorrect” was 64% less frequent in recovered reflections (1.1 vs. 3.1, $p < 0.001$): unrecovered reflections more often restated that something was incorrect, failed, or missing without specifying what should be checked or changed next. Representative cases followed the same pattern. High-capability reflections identified specific technical mismatches, such as the need to coordinate pod ports and service `targetPort` values, and proposed explicit end-to-end validation steps. Low-capability reflections tended to acknowledge that the diagnosis was unclear but did not identify the wrong command, the overlooked evidence, or the concrete repair to try next.

Categorizing the evaluator’s failure reason for the 310 failed first attempts that triggered these reflections yields a coarse taxonomy of first-attempt failure modes: the pod was not running or not ready in 49.4% of cases, the service was unreachable on its advertised port in 41.0%, the agent had deleted a resource without recreating it in 4.8%, and the service had no matching endpoints in 3.2% (remaining 1.6% other). The third category is a pure process failure: the agent’s destructive repair step, not the original fault, caused

Recovered reflection (action-oriented)	Unrecovered reflection (vague acknowledgment)
“I corrected the pod’s <code>containerPort</code> to 8765, but the Service still targets the previous port, so traffic never reaches the container. The fix was incomplete: I repaired one side of the port mapping and did not verify end-to-end reachability. Next attempt: patch the Service <code>targetPort</code> to 8765 and confirm the endpoint responds before stopping.”	“The pod is still not ready and the previous fix did not resolve the problem. Something in the configuration is incorrect and the service is missing an endpoint. I should re-examine the pod and try a different change next time.”
<i>Representative reflections illustrating the patterns quantified in Table 18: reflections followed by a successful second attempt perform explicit credit assignment and name concrete verification steps (“verify”/“test” terms appear markedly more often), whereas unrecovered reflections restate the symptom without identifying the wrong action or the next repair. Verbatim logs are available in the replication database.</i>	

FIGURE 5. Representative self-reflection contrast.

the failing state that the subsequent reflection had to diagnose.

K. WORKED EXAMPLE: ONE REFLECTION LOOP

As an illustration, we trace a single full-Reflexion run on Case 1 (wrong port) with Claude Haiku 4.5. In this incident the pod runs but is unreachable through its Service because of a port mismatch. The run reached success after two trials in 168.6 s at a total cost of \$0.094.

Trial 1 (diagnosis and partial repair). The Actor executed its diagnosis phase (`kubectl get, describe, logs, get events`) and found that the pod declared one port while the application listened on another. It deleted and recreated the pod with the corrected `containerPort`. The deterministic Evaluator, however, returned failure: the endpoint was still unreachable, with reason “Service ‘wrong-port-service’ not responding on port 80.” The repair had addressed the pod side of the mapping but left the Service targeting the old port.

Reflection. The Self-Reflection module analyzed the failed trajectory and the Evaluator reason and produced a first-person reflection that (i) assigned credit to the incomplete fix, (ii) identified the unrepaired Service `targetPort` as the root cause, and (iii) prescribed a concrete next step:

“I fixed the pod’s container port but did not update the Service, so traffic is routed to the wrong port. Next time I should patch the Service `targetPort` to match the container and confirm reachability before stopping.”

Trial 2 (guided repair and success). Conditioned on this reflection from episodic memory, the Actor targeted the Service directly, aligned its `targetPort` with the container, and stopped with `FIX COMPLETE`. The Evaluator then confirmed that pods were running, containers ready, and the endpoint reachable, marking the run successful. This single run shows the paper’s central finding on a small scale: the second attempt provides most of the value, and reflection contributes when the first attempt leaves a partially repaired, still-inconsistent state that must be reasoned about rather than merely retried.

V. DISCUSSION

This section discusses four themes: the value of retrying, the conditional nature of reflection, the moderating role of

model capability, and the operational implications for AIOps deployments. It closes with the threats to validity.

A. THE DOMINANT VALUE OF RETRYING

The main aggregate result is that an additional, externally verified attempt accounts for most of the improvement. The second attempt improved success by 29.8 percentage points, far exceeding the overall incremental gain from explicit reflection. This implies that LLM agents often gather useful evidence during the first attempt, and a further chance to act can correct many errors even without a separate reflection step.

For system designers, this finding suggests that agentic remediation systems should prioritize reliable retry and validation loops before adding more elaborate self-improvement mechanisms. A deterministic evaluator is particularly important because it stops the agent from prematurely claiming success and ensures that retries are driven by external evidence rather than the model’s own judgment. The self-verification ablation (Section IV-G) supports this point: letting the Actor verify and revise within one attempt recovered about half of the retry effect on single-fault scenarios, but only for the stronger models, largely by waiting for the cluster to converge, and at more than twice the baseline cost; a misleading local check also led Haiku 4.5 to accept a broken service in every Case 4 run. An external evaluator that probes the service from outside the pod remains the more reliable and cheaper source of the verification signal.

This interpretation is consistent with the broader agent literature, where tool-using LLM systems are usually most effective when reasoning is coupled with environmental feedback rather than treated as a single-pass generation problem [5], [6], [14]. In Kubernetes troubleshooting, the first attempt often functions as an exploratory interaction: commands reveal pod events, service endpoints, logs, and configuration mismatches that were not available in the initial prompt. The observed jump from baseline to two-try therefore suggests that verified interaction itself is a major source of performance improvement, aligning with ReAct-style findings that alternating between action and observation can improve task completion. Our results also qualify that literature: an additional action cycle is not the same as successful self-reflection, because the evaluator-driven retry produced most of the aggregate gain without a separate reflective prompt.

B. REFLECTION IS CONDITIONAL, NOT UNIVERSAL

On its own, the aggregate Reflexion gain of 0.8 percentage points is small in practical terms, but the average hides a preliminary dependence on problem complexity. Reflection slightly reduced performance in single-fault scenarios but improved results in the two tested multi-fault scenarios. Because the double-fault and triple-fault categories each contain one scenario, this pattern should be read as hypothesis-generating rather than definitive. A likely explanation is that single-fault cases often require a direct correction, and extra reflection may introduce irrelevant context or distract the model. Multi-fault cases instead require the agent to revise its hypothesis after a partial repair. In those settings, explicit credit assignment and concrete next actions may help prevent the agent from repeating an incomplete strategy.

The feedback-only control (Section IV-F) narrows this interpretation. On the multi-fault scenarios, supplying the evaluator's verbatim failure reason without any reflection outperformed full Reflexion on the three models tested. The multi-fault benefit previously attributed to reflection is therefore better described as a benefit of *access to a concrete external failure signal*: the verbal reflection stage adds cost and latency, and, at least as implemented here without direct access to the evaluator's textual reason, it transmits that signal less faithfully than the raw feedback itself. The design recommendation changes accordingly: before adding a reflection module, a remediation system should first make sure that whatever failure evidence the evaluator produces reaches the next attempt verbatim. Reflection then becomes a candidate mechanism for the residual cases in which no informative external signal exists and the agent must reconstruct the failure from its own trajectory.

This result supports complexity-aware orchestration. A production-oriented AIOps system does not need to call reflection for every incident. Reflection can instead be activated after a failed first attempt, when several resources show inconsistent states, or when the evaluator observes that a partial fix changed the symptom without resolving it.

The complexity-dependent pattern is consistent with prior work on Reflexion and self-refinement but also qualifies it. Reflexion reports that verbal feedback can help agents convert failed trajectories into improved subsequent behavior, and Self-Refine similarly shows that critique-and-revision can improve outputs without parameter updates [2], [4]. Our Kubernetes results provide preliminary support for this principle in the tested multi-fault incidents, where the first attempt may repair only part of the causal chain and reflection can summarize what remains unresolved. However, the slight decline in single-fault cases indicates that reflection is not automatically beneficial: when the correct action is local and direct, additional natural-language analysis may increase cognitive load, introduce irrelevant hypotheses, or bias the model away from a simple fix. Reflection is therefore best understood as a targeted credit-assignment mechanism for stateful, multi-step failures, not as a universal performance enhancer.

C. MODEL CAPABILITY MODERATES REFLECTION

Within the five Claude models evaluated, the link between model capability and reflection effectiveness is non-linear. Because all models come from a single family, capability here is confounded with version-specific training and tool-use behavior, so the pattern is within-family evidence, not a general rule across model families. Lower-capability models appear to generate or use reflections poorly, so the added context does not translate into better actions. Mid-capability models can reason over the feedback and still have meaningful room for improvement. Higher-capability models solve many incidents without reflection, creating a ceiling effect. This pattern indicates that reflection cannot replace sufficient model capability; it appears most useful when the model can understand the feedback but is not already strong enough to succeed with a simple retry.

This non-linear effect also bears on recent AIOps and LLM-for-operations studies. Prior AIOps research emphasizes that operational automation depends not only on model intelligence but also on telemetry quality, action constraints, and feedback loops [10], [24]. Our findings are consistent with that view: reflection helped when the model had enough capability to map evaluator feedback to concrete Kubernetes actions, but it did not compensate for weak tool-use or incomplete domain reasoning. The result also suggests that benchmark reports should separate model capability, retry budget, and reflection design, because combining them into a single success number can overstate the independent contribution of reflective memory.

D. OPERATIONAL IMPLICATIONS

The findings favor a tiered remediation policy. For straightforward incidents, a low-latency two-try setup may be sufficient. For complex or unresolved incidents, the system can escalate to Reflexion with a mid-capability model. For high-severity incidents where success outweighs cost, a stronger model such as Opus may be appropriate, although reflection may add little at that level. In every case, remediation should remain verification-driven, with a programmatic evaluator deciding whether the agent continues, reflects, escalates, or stops.

In an enterprise environment, the policy's inputs and outputs would connect to existing systems. The evaluator's failure signal can come from the observability and alerting platforms already in place instead of custom probes, and the repairs an agent proposes can be routed through GitOps- or CI/CD-style review pipelines so that cluster changes remain declarative, versioned, and approvable instead of being applied imperatively. Connecting the escalation step to incident-management tooling would let unresolved cases be handed to on-call engineers with the full diagnostic trajectory attached.

Algorithm 1 summarizes this policy.

E. THREATS TO VALIDITY

Several validity threats remain. First, LLM behavior is non-deterministic. We addressed this by repeating each condition ten times, but more repetitions would narrow the confidence

Algorithm 1 Proposed tiered remediation policy derived from the findings (deployment recommendation; not the experimental protocol of Section III).

Require: incident I , model tier ladder $M_1 \prec M_2 \prec \dots \prec M_k$

Require: deterministic evaluator E , attempt budget B

- 1: $m \leftarrow M_1$ {start with a fast, low-cost model}
- 2: $r_1 \leftarrow \text{Actor}(I, m)$ {one-shot remediation attempt}
- 3: **if** $E(r_1) = \text{success}$ **then**
- 4: **return** success {most incidents resolve on a verified first/second try}
- 5: **end if**
- 6: $s \leftarrow E.\text{signal}(r_1)$ {partial progress / multi-resource inconsistency?}
- 7: **if** attempts $< B$ **then**
- 8: **if** s indicates a partial or multi-fault state **then**
- 9: $\rho \leftarrow \text{Reflect}(r_1, s)$ {credit assignment + next steps}
- 10: $r_2 \leftarrow \text{Actor}(I, m, \rho)$ {reflection-guided retry}
- 11: **else**
- 12: $r_2 \leftarrow \text{Actor}(I, m)$ {plain verified retry; no reflection}
- 13: **end if**
- 14: **if** $E(r_2) = \text{success}$ **then**
- 15: **return** success
- 16: **end if**
- 17: **end if**
- 18: **if** severity high **and** cost budget allows **then**
- 19: escalate to a stronger tier $m \leftarrow M_{j>1}$ and repeat
- 20: **else**
- 21: escalate to a human operator {dry-run, approve, or edit the repair}
- 22: **end if**

intervals. Second, the benchmark includes ten scenarios and therefore cannot represent the full range of production Kubernetes incidents. The complexity analysis is especially limited because only two scenarios are multi-fault cases; consequently, the observed gains for double- and triple-fault incidents should be treated as evidence of a promising pattern rather than as a definitive general result. Third, all runs used a single-node Minikube v1.32 environment on Docker Desktop; multi-node clusters could introduce scheduling, networking, storage, and permission problems not present here. Fourth, the evaluation covered only Claude models because the Actor was implemented with the Claude Agent SDK. Whether the findings generalize to GPT, Gemini, LLaMA, or other model families remains open. Fifth, success was measured as a binary outcome. An agent that found the root cause but failed to apply the repair was scored the same as one that made an unrelated change. Future work should use graded metrics for diagnostic accuracy, action safety, repeated mistakes, rollback behavior, and partial remediation progress. Sixth, the Actor is prohibited from verifying its own fix because verification is delegated to the deterministic Evaluator. This design choice may inflate the measured value

of the second attempt: the baseline effectively measures a single unverified attempt, whereas the retry conditions add both another attempt and an external verification signal, so part of the 29.8-percentage-point retry effect may reflect the value of verification itself rather than retrying alone. The post-hoc self-verification ablation (Section IV-G) bounds this concern rather than removing it: on the single-fault scenarios and the three current models, permitting in-attempt verification recovered about half of the retry effect (+13.3 of +25.0 percentage points), most of it in one scenario where the baseline failure was a convergence-timing artifact, and the externally verified second attempt remained significantly better. The ablation uses five repetitions per cell, excludes the two retired models, and does not cover the multi-fault scenarios where the reflection benefit was concentrated; the retry effect should therefore be interpreted as the value of an additional, externally verified attempt, of which self-verification alone captures only part. Seventh, the fault patterns in the eight single-fault scenarios are pervasive in tutorials, community forums, and public benchmarks, so descriptions of these misconfigurations and their fixes plausibly appear in the models' training data. Absolute success rates should therefore be read as an upper bound on performance for genuinely novel incidents rather than as an estimate of general troubleshooting ability. The comparisons between configurations, however, remain internally valid: the baseline, retry, and Reflexion conditions used identical scenarios, models, and prompts, so any familiarity with a fault pattern is shared equally by all conditions and cannot by itself produce the observed between-condition differences. In addition, the two multi-fault scenarios, where the reflection benefit is concentrated, were composed for this study by combining faults across resources, making verbatim memorization of their solutions unlikely, although the constituent single-fault patterns remain individually well known. Eighth, the feedback-only control was added post hoc and could be run only on the three models still available through the API, on the multi-fault scenarios; its comparison against the other arms uses only matched runs executed on the identical current scenario instantiation, which differs from the instantiation of the original five-model matrix in its diagnostic entry point (Section IV-F), and the control receives the evaluator's textual reason, which the implemented reflection pipeline does not. Its results should therefore be read as isolating the value of the evaluator's failure signal in the tested cells, not as a full factorial re-run of the study.

VI. CONCLUSION AND FUTURE WORK

This study assessed Reflexion for autonomous Kubernetes incident remediation using 1,500 controlled experiments. The results show that allowing a second attempt substantially improves LLM-agent success, whereas explicit self-reflection adds only a small aggregate benefit. Its value is most visible in specific settings: the two tested multi-fault incidents and mid-capability models. Because the multi-fault evidence rests on only two scenarios, this complexity-specific benefit remains preliminary and requires confirmation on a

broader multi-fault benchmark. The post-hoc feedback-only control further shows that, on identically instantiated multi-fault scenarios, forwarding the evaluator's verbatim failure reason to the second attempt is clearly more effective than verbal self-reflection, so most of the observed benefit is due to the failure signal itself. Lower-capability models do not exploit reflection well, while higher-capability models often succeed without it. The cost and latency analyses suggest that self-reflection should be used selectively, together with deterministic evaluation and dynamic model selection. More broadly, self-reflective LLM agents can support autonomous operations, but their design should account for incident complexity, model capability, and verification requirements. This conclusion concerns single-round reflection under a two-attempt budget; whether reflection's value grows under larger budgets, where episodic memory can accumulate across trials, remains an open question.

In direct response to the research questions, **RQ1** shows that single-round Reflexion adds only a small aggregate gain beyond a verified retry (69.2% versus 68.4%); the self-verification ablation further indicates that roughly half of the retry effect on single-fault scenarios reflects verification rather than the extra attempt itself, with the externally verified retry still superior. **RQ2** shows a preliminary signal of benefit in the two multi-fault scenarios, but this result requires more double- and triple-fault cases; the feedback-only control further indicates that this benefit is mediated by access to the evaluator's failure signal rather than by the reflection step itself. **RQ3** shows that, within the Claude family evaluated, mid-capability models benefit most from reflection, while weak models underuse it and the strongest model reaches a ceiling; confirming this across other model families remains future work. **RQ4** indicates that Haiku 4.5 provides the best observed success–cost trade-off among the retry-capable settings, whereas Opus 4.5 provides the highest absolute success when cost is less important.

Two conclusions follow. First, Reflexion should not be interpreted as a general replacement for careful agent design, reliable tool access, or external validation. In line with prior work on ReAct, Reflexion, and tool-using LLM agents, the strongest improvements arise when the model is embedded in a loop that exposes observations, checks outcomes, and constrains the next action [4]–[6]. Second, the results show that self-reflection has practical value when it is matched to the structure of the task. Kubernetes incidents with several interacting faults create a need for causal revision and partial-progress tracking, and this is where an informative external failure signal most clearly helps; the feedback-only control shows that delivering that signal verbatim to the next attempt can be more effective than reflecting on it. For simpler incidents, a verified retry alone can provide most of the benefit with lower cost and latency, and self-reflection remains a candidate mechanism mainly for settings in which no informative external signal is available.

For AIOps practice, the study supports selective, evidence-driven deployment. Rather than enabling reflection for every

alert, an operational system can first run a fast remediation attempt, evaluate the cluster state programmatically, and then decide whether to retry, reflect, escalate to a stronger model, or involve a human operator. This conclusion is consistent with AIOps literature that emphasizes closed-loop automation, telemetry quality, and safety-aware decision making over unconstrained model autonomy [10], [24]. The practical implication is that future production systems should treat reflection as one component in a broader remediation pipeline, not as the sole source of reliability.

Future studies should examine additional LLM families, larger and more varied Kubernetes benchmarks, multi-node production-like clusters, and remediation processes with more than two cycles. Human-in-the-loop remediation is also important, especially for high-risk operations where an agent proposes a repair and a human operator approves or edits it before execution. Safety constraints require further study as well: autonomous remediation agents can delete pods, patch services, and modify deployments, so production use should be controlled through permissions, dry-run execution, rollback plans, audit logs, and policy checks; recent work on trustworthy LLM agents catalogues these threats and countermeasures systematically [31]. The architecture used here already records every command and evaluator verdict per run, which provides a natural audit trail; production readiness would further require explicit approval workflows for destructive actions and automated policy-compliance checks integrated with the organization's change-management process.

Future work should also move beyond binary success and failure labels. More detailed metrics could distinguish correct diagnosis, safe but incomplete repair, unsafe action, unnecessary change, rollback success, and time-to-stable-state. Such metrics would make it possible to evaluate whether reflection improves the agent's understanding of the root cause even when the final repair is incomplete. Because the public replication dataset stores the full command trajectory, evaluator reason, and token accounting for every run, metrics such as root-cause localization accuracy, repair precision, and action efficiency can be computed retrospectively from the released artifacts without rerunning the experiments. Another important direction is to analyze the content of generated reflections themselves: reflections may be useful when they identify overlooked evidence and concrete next actions, but harmful when they restate symptoms, invent unsupported causes, or overfit to misleading intermediate observations. Because each scenario was repeated ten times, the released corpus of 310 reflections also enables a retrospective study of the semantic consistency, diversity, and quality of reflections generated across repeated runs of identical incidents.

Finally, future research should investigate adaptive orchestration policies. A promising direction is to trigger reflection only when evaluator signals indicate partial progress, repeated failure, or multi-resource inconsistency, and to combine this trigger with dynamic model selection based on incident severity and expected cost. The tiered policy in AI-

gorithm 1 is a first step in this direction; confidence-aware or uncertainty-driven triggers that estimate when reflection is worth its added cost and latency would be a natural refinement. Larger benchmarks should include network policies, persistent volumes, RBAC errors, node failures, autoscaling behavior, and multi-service dependencies, as well as Helm-based deployments, StatefulSets, admission controllers, and heterogeneous multi-node workloads, so that results better approximate production Kubernetes environments. Evaluating these settings across multiple LLM providers would clarify whether the conditional value of reflection observed here is a general property of agentic remediation or a model-family-specific effect.

ACKNOWLEDGMENT

The authors thank the anonymous reviewers for their constructive comments, which substantially improved the manuscript.

DATA AND CODE AVAILABILITY

The experimental outputs were saved as per-run JSON files and exported into an SQLite database. The complete replication package (raw per-run JSON records, the SQLite results database, all agent and reflection prompts, the scenario definitions with automated setup and cleanup scripts, and the programmatic evaluator logic) is publicly available at <https://github.com/mmerts/thesis-sdk> (accessed August 13, 2026). The exact API model identifiers used in all experiments were `claude-3-haiku-20240307`, `claude-3-5-haiku-20241022`, `claude-haiku-4-5-20251001`, `claude-sonnet-4-5-20250929`, and `claude-opus-4-5-20251101` for Claude 3 Haiku, Claude 3.5 Haiku, Claude Haiku 4.5, Claude Sonnet 4.5, and Claude Opus 4.5, respectively.

REFERENCES

- [1] S. Ghosh, M. Shetty, C. Bansal, and S. Nath, "How to Fight Production Incidents? An Empirical Study on a Large-Scale Cloud Service," in *Proc. 13th ACM Symposium on Cloud Computing (SoCC)*, 2022, pp. 126–141.
- [2] A. Madaan, N. Tandon, P. Gupta, S. Hallinan, L. Gao, S. Wiegrefe, U. Alon *et al.*, "Self-Refine: Iterative Refinement with Self-Feedback," in *Advances in Neural Information Processing Systems*, vol. 36, 2023.
- [3] S. Yao, D. Yu, J. Zhao, I. Shafraan, T. L. Griffiths, Y. Cao, and K. Narasimhan, "Tree of Thoughts: Deliberate Problem Solving with Large Language Models," in *Advances in Neural Information Processing Systems*, vol. 36, 2023.
- [4] N. Shinn, F. Cassano, E. Berman, A. Gopinath, K. Narasimhan, and S. Yao, "Reflexion: Language Agents with Verbal Reinforcement Learning," in *Advances in Neural Information Processing Systems*, vol. 36, 2023.
- [5] L. Wang, C. Ma, X. Feng, Z. Zhang, H. Yang, J. Zhang, Z. Chen, J. Tang, X. Chen, Y. Lin, W. X. Zhao, Z. Wei, and J.-R. Wen, "A Survey on Large Language Model Based Autonomous Agents," *Frontiers of Computer Science*, vol. 18, no. 6, p. 186345, 2024.
- [6] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafraan, K. Narasimhan, and Y. Cao, "ReAct: Synergizing Reasoning and Acting in Language Models," in *Proc. International Conference on Learning Representations (ICLR)*, 2023.
- [7] J. S. Park, J. C. O'Brien, C. J. Cai, M. R. Morris, P. Liang, and M. S. Bernstein, "Generative Agents: Interactive Simulacra of Human Behavior," in *Proc. 36th Annual ACM Symposium on User Interface Software and Technology (UIST)*, 2023, pp. 1–22.
- [8] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes: Lessons Learned from Three Container-Management Systems Over a Decade," *Queue*, vol. 14, no. 1, pp. 70–93, 2016.
- [9] Cloud Native Computing Foundation, "Kubernetes Documentation," Online, accessed: Aug. 18, 2026. [Online]. Available: <https://kubernetes.io/docs/>
- [10] L. Zhang, T. Jia, M. Jia, Y. Wu, A. Liu, Y. Yang, Z. Wu, X. Hu, P. S. Yu, and Y. Li, "A Survey of AIOps for Failure Management in the Era of Large Language Models," arXiv:2406.11213, 2024.
- [11] S. He, P. He, Z. Chen, T. Yang, Y. Su, and M. R. Lyu, "A Survey on Automated Log Analysis for Reliability Engineering," *ACM Computing Surveys*, vol. 54, no. 6, pp. 130:1–130:37, 2021.
- [12] A. Jones, "K8sgpt: Kubernetes Diagnostics with AI," Computer software, 2023, accessed: Aug. 18, 2026. [Online]. Available: <https://github.com/k8sgpt-ai/k8sgpt>
- [13] M. S. Ardebili and A. Bartolini, "KubeIntellect: A Modular LLM-Orchestrated Agent Framework for End-to-End Kubernetes Management," *Journal of Grid Computing*, vol. 24, no. 3, p. 17, 2026.
- [14] Y. Qin, S. Hu, Y. Lin, W. Chen, N. Ding, G. Cui, Z. Zeng *et al.*, "Tool Learning with Foundation Models," *ACM Computing Surveys*, vol. 57, no. 4, pp. 1–40, 2024.
- [15] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu *et al.*, "AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation," in *Proc. First Conference on Language Modeling (COLM)*, 2024.
- [16] T. Schick, J. Dwivedi-Yu, R. Dessi, R. Raileanu, M. Lomeli, L. Zettlemoyer, N. Cancedda, and T. Scialom, "Toolformer: Language Models Can Teach Themselves to Use Tools," in *Advances in Neural Information Processing Systems*, vol. 36, 2023.
- [17] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de O. Pinto, J. Kaplan *et al.*, "Evaluating Large Language Models Trained on Code," arXiv:2107.03374, 2021.
- [18] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?" in *Proc. International Conference on Learning Representations (ICLR)*, 2024.
- [19] X. Liu, H. Yu, H. Zhang, Y. Xu *et al.*, "AgentBench: Evaluating LLMs as Agents," in *Proc. International Conference on Learning Representations (ICLR)*, 2024.
- [20] M. D. Jesus, P. Sylvester, W. Clifford, A. Perez, and P. Lama, "LLM-Based Multi-Agent Framework for Troubleshooting Distributed Systems," in *Proc. IEEE Cloud Summit*, 2025, pp. 110–115.
- [21] Y. Chen, M. Shetty, G. Somashekar, M. Ma, Y. Simmhan, J. Mace, C. Bansal, R. Wang, and S. Rajmohan, "AIOpsLab: A Holistic Framework to Evaluate AI Agents for Enabling Autonomous Clouds," in *Proc. Eighth Annual Conference on Machine Learning and Systems (MLSys)*, 2025.
- [22] S. Jha, R. R. Arora, Y. Watanabe, T. Yanagawa, Y. Chen *et al.*, "ITBench: Evaluating AI Agents across Diverse Real-World IT Automation Tasks," in *Proc. 42nd International Conference on Machine Learning (ICML)*, PMLR vol. 267, 2025, pp. 27 134–27 197.
- [23] J. D. de Arcaya, A.-I. Torre-Bastida, G. Zarate, R. M. n n, and A. Almeida, "A Joint Study of the Challenges, Opportunities, and Roadmap of MLOps and AIOps: A Systematic Survey," *ACM Computing Surveys*, vol. 56, no. 4, pp. 1–30, 2023.
- [24] L. Zhang, T. Jia, M. Jia, Y. Wu, A. Liu, Y. Yang, Z. Wu, X. Hu, P. S. Yu, and Y. Li, "A Survey of AIOps in the Era of Large Language Models," *ACM Computing Surveys*, vol. 58, no. 2, pp. 1–35, 2025.
- [25] J. Zha, X. Shan, J. Lu, J. Zhu, and Z. Liu, "Leveraging Large Language Models for Efficient Alert Aggregation in AIOps," *Electronics*, vol. 13, no. 22, p. 4425, 2024.
- [26] Y. Zhou, Y. Chen, X. Rao, Y. Zhou, Y. Li, and C. Hu, "Leveraging Large Language Models and BERT for Log Parsing and Anomaly Detection," *Mathematics*, vol. 12, no. 17, p. 2758, 2024.
- [27] L. Ma, W. Yang, Y. Li, B. Fei, M. Zhou, S. Li, S. Jiang, B. Xu, and Y. Xiao, "AdaptiveLog: An Adaptive Log Analysis Framework with the Collaboration of Large and Small Language Model," *ACM Transactions on Software Engineering and Methodology*, vol. 35, no. 5, pp. 1–42, 2026.
- [28] Y. Gao, T. Luo, K. Huang, J. Tang, and X. Li, "LogLAA: An Adaptive Integrated Log Anomaly Analysis Framework," *Cybersecurity*, vol. 9, p. 141, 2026.
- [29] M. M. Süerkan, "AI Agents for Kubernetes Incident Remediation: Evaluating the Reflexion Framework," Master's thesis, TED University, Graduate School, Computer Engineering, Ankara, Türkiye, 2026.

- [30] Anthropic, “Claude Agent SDK Overview,” Online, 2024, accessed: Aug. 18, 2026. [Online]. Available: <https://platform.claude.com/docs/en/agent-sdk/overview>
- [31] M. Yu, F. Meng, X. Zhou, S. Wang, J. Mao, L. Pang *et al.*, “A Survey on Trustworthy LLM Agents: Threats and Countermeasures,” in *Proc. 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, 2025, pp. 6216–6226.



MUSTAFA MERT SUERKAN is a graduate student at TED University, Ankara, Türkiye. His research interests include large language model agents, autonomous cloud operations, Kubernetes troubleshooting, AIOps, and software engineering automation.



EMIN KUGU (Member, IEEE) received the B.S. degree from Istanbul University Computer Science Engineering Department, his master's degree from Air Force Academy Software Engineering and his doctoral degree from Old Dominion University Electrical and Computer Engineering program. He worked as a part-time lecturer at different universities. He worked as a project manager in the Air Force Command Information System (HvBS) project and took part in the software development

processes of international defense projects. His research interests include cyber security, computer networks, artificial intelligence and machine/deep learning.



HAKAN EMEKCI (Member, IEEE) received the B.Sc. degree in computer engineering from Middle East Technical University (METU), Ankara, Türkiye, in 2011, the M.Sc. degree from London Business School, and the Ph.D. degree from Hacettepe University, Ankara, Türkiye. He is currently a Faculty Member with Hacettepe University, specializing in artificial intelligence. He is also the Founder of NavigaAI, a company focused on AI-driven solutions. He is actively involved

in educational technology development. He has led impactful projects in collaboration with the European Bank for Reconstruction and Development (EBRD) and the United Nations Development Program (UNDP), where his work has supported AI-driven solutions for social and economic development initiatives. Beyond his academic and professional achievements, he has been dedicated to bridging advanced AI methodologies with practical applications that address real-world challenges, advancing both the technology and its accessibility. He continues to mentor the next generation of data scientists, focusing on ethical AI practices and innovative problem-solving approaches.

...