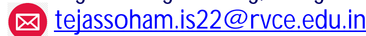
IRJCS: International Research Journal of Computer Science
Volume 13, Issue 05, May 2026
ISSN: 2393-9842
https://www.irjcs.com/archives
https://doi.org/10.26562/irjcs.2025.v1305.04

Argo AI: A Multi-Agentic RAG System for GitOps

Tejas Soham
Dept. of Information Science and Engineering,
RV College of Engineering, Bengaluru, India

tejassoham.is22@rvce.edu.in
ROR https://rvce.edu.in/
https://orcid.org/0009-0004-9398-7488

Rekha B.S
Dept. of Information Science and Engineering,
RV College of Engineering, Bengaluru, India

rekhabs@rvce.edu.in
ROR https://rvce.edu.in/
https://orcid.org/0000-0002-2616-6725

Publication History:
Manuscript Reference No: IRJCS/RS/Vol.13/Issue05/MYCS10092
Research Article | Open Access | Double-Blind Peer-Reviewed | Article ID: IRJCS/ RS/ Vol.13/ Issue 05/ CSMY26. MYCS10092 Received: 24, April 2025, Revised: 04, May 2026, Accepted: 16, May 2026, Published Online: 20, May 2026.
https://www.irjcs.com/volumes/Vol13/iss-05/04.MYCS10092.pdf
Article Citation: Tejas,Rekha(2026),Argo AI: A Multi-Agentic RAG System for GitOps, IRJCS: International Research Journal of Computer Science, Volume 13, Issue 04 of 2026 pages 584-591
Doi:> https://doi.org/10.26562/irjcs.2026.v1305.04 ROR https://rvce.edu.in/
BibTeX Key: Tejas@2026Agro AI. IRJCS papers should be cited as IRJCS (International Research Journal of Computer Science, AM Publications, India 2026, ISSN 2393-9842, https://doi.org/10.26562/irjcs.2026.v1305.04. The journal's official abbreviation is IRJCS. ORCiD: https://orcid.org/0009-0004-9398-7488 About the License: Copyright©2026 copyright by the authors. This article is an open access and license under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

Abstract: Argo CD reconciles Kubernetes cluster state against Git repositories, but it reports deployment failures without explaining their cause. Current diagnostic tools for Kubernetes either lack awareness of Argo CD application states, process telemetry through a single LLM prompt without iterative reasoning, or respond to metric alerts without parsing GitOps configuration signals. Argo AI introduces a heuristic A2A-style router that dispatches incoming cluster telemetry to five specialist agents (Runtime, Config, Network, Storage, RBAC) by matching deterministic Kubernetes pod state strings and event reason fields, removing LLM token cost from routing entirely. The system's two-pod security architecture confines the Python reasoning layer to zero Kubernetes RBAC permissions, forcing every cluster query through a read-only Go proxy and blocking prompt injection from reaching the Kubernetes API. SHA-256 fingerprint caching paired with regex-based log pre- filtering reduces repeated diagnosis latency by 99.57% (from8.57sto 36.8ms) and cuts the average token payload by 87.08%. Tested on seven injected failure scenarios, the router achieves 100% dispatch accuracy, and a FAISS-backed retrieval pipeline over 4,801 documentation chunks reaches 96.0% accuracy at a similarity threshold of 0.65. Argo AI outputs copyable Git patch suggestions; it never modifies the cluster.
Index Terms: Argo CD, Kubernetes, GitOps, multi-agent systems, large language models, root cause analysis, retrieval-augmented generation

INTRODUCTION

Production container workloads now run on Kubernetes at a scale where manual state management is infeasible [1]. Teams that adopt Kubernetes commit their desired cluster state to a Gitrepository and delegate the job of applying it to be conciliation controller running inside the cluster, a workflow known as GitOps [2]. ArgoCD is the dominant reconciliation controller in this space [3]: it polls one or more Git repositories, computes diffs between declared manifests and live objects, and pushes the cluster toward the declared state whenever drift is detected. Argo CD handles the normal case well: pull a commit, apply manifests, and report success. Failure diagnosis is different. When an application enters Degraded, Out of Sync, Missing, or Error state, Argo CD reports the transition but provides no root cause. The engineer must then separately inspect pod logs, Kubernetes events, YAML diffs, RBAC policies, and storage bindings, each through a different kubectl or CLI command. Multi-resource failures compound this triage time because each failing object can mask or depend on another. Argo AI was built to close this diagnostic gap. Three specific problems shaped its architecture. Routing a diagnostic query to the correct reasoning con-text costs tokens when done through an LLM, because a classifier model must run inference on every request just to select the right agent. Argo AI side steps this cost with a heuristic router that matches on Kubernetes pod state strings (OOM Killed, Crash Loop Back Off) and event reason fields (Failed Mount, Unhealthy). The Kubernetes control plane generates these strings deterministically, making pattern matching a reliable routing signal with zero model overhead. Giving an LLM direct access to the Kubernetes API creates a prompt injection risk. If the model can be tricked into executing arbitrary API calls, a crafted input could escalate privileges or exfiltrate secrets. Argo AI blocks this attack surface by splitting the system into two pods: a Go service that holds the Kubernetes Service Account and serves read- only data, and a Python agent with zero RBAC permissions that reaches the cluster only through the Go service's internal HTTP interface. Raw container logs are dominated by health check pings, initialization traces, and heartbeat lines.

Sending thousands of repetitive tokens to the LLM wastes context window capacity and inflates API cost, so Argo AI pre-filters logs with a regex that keeps only error-level keywords, cutting the average token count by 87% (Table III).

The contributions of this paper are:

1) A heuristic A2A-style routing mechanism that dispatches incoming telemetry to five domain-specialist agents by matching Kubernetes signal patterns, achieving 100% routing accuracy across seven failure types without consuming LLM tokens on the routing decision (Table I).
2) The two-service security architecture that isolates the AI reasoning layer from Kubernetes credentials, preventing prompt injection attacks from reaching the cluster API.
3) SHA-256 fingerprint caching paired with regex-based log pre-filtering that reduces repeated diagnosis latency by 99.57% (Table II) and token payload by 87.08% (Table III).

II. RELATED WORK

A. AI Assistants for Kubernetes

Open Shift Light speed (OLS)[4] combines Lang Chain, Llama Index, and a FAISS vector store into a documentation-oriented assistant for OpenShift and Kubernetes operators. Its scope is bounded to answering questions against indexed docs; it cannot parse Argo CD application states, sync conditions, or desired-vs-live configuration deltas. Cluster telemetry reaches OLS only through external Model Context Protocol servers, which positions it as a reactive question-answering tool rather than an active diagnostic agent. The Assistant for Argo CD [5] is a Type Script UI extension that adds a chat panel to the Argo CD dashboard, feeding manifests, events, and logs into a single LLM prompt via llama-stack. Its single-pass architecture has no mechanism for iterative reasoning, tool execution, or retrieval augmentation. Large logs and resource files exceed the prompt window quickly, limiting the system to small-payload cases. OpenClaw [6] is a Go-based Agentic diagnostics platform from RedHat that runs CLI commands inside isolated and box pods. It reacts to Prometheus alerts but has no GitOps awareness: it cannot parse Argo CD Application resources, evaluate sync states, or suggest Git-formatted fixes. Argo AI is the first tool in this space that routes on Argo CD- native signal patterns (pod state reasons and event reasons) rather than free-text classification, and the first that physically isolates the reasoning pod from Kubernetes RBAC.

B. Multi-Agent LLM Systems

KubeIntellect [7] showed that modular multi-agent pipelines outperform single monolithic models at tool execution tasks in Kubernetes environments. STRATUS [8] introduced a transactional no-regression specification for SRE agent actions at Neur IPS 2025, showing that state-machine-organized agents with rollback capability prevent cascading errors during autonomous cloud remediation. TAMO [9] decouple draw multimodal telemetry from LLM reasoning by routing logs, metrics, and traces through specialized alignment and localization tools before passing structured context to the model. Google's Agent Development Kit (ADK) [10] provides the agent construction primitives used in Argo AI: Llm Agent classes, Function Tool wrappers, and session-scoped reasoning loops. Lite LLM [11] abstracts the API differences between providers (Gemini, Open AI, Anthropic, Ollama) behind a single interface, enabling Argo AI's Bring-Your-Own-Model capability. Argo AI's router spends zero LLM tokens because it matches on deterministic Kubernetes strings rather than running a learned classifier. Its security model is distinct from the systems above: the reasoning pod and the cluster-access pod run as separate Kubernetes deployments with independent RBAC bindings.

C. AI Ops and Incident Diagnosis

Earlier AI Ops work established that automating root cause analysis shortens Mean Time to Resolution (MTTR) in cloud deployments [12], [13]. RCA Copilot [14] paired LLMs with structured incident handler outputs and demonstrated higher diagnostic accuracy than rule-based systems alone at Microsoft scale. Mutiny! [15] Analyzed Kubernetes failure patterns at IEEE DSN 2024 and found that a large fraction of outages stem from configuration dependencies between resources, con-firming the need for tools that trace cross-resource relation- ships. Argo AI applies these findings to the Argo CD domain specifically by combining LLM reasoning with Argo CD signal collection and accurate documentation index of 4,801 chunks.

III. SYSTEM DESIGN

A. Overview

Argo AI runs as two Kubernetes pods inside the Argo CD agent namespace. Fig. 1 shows the production layout. When the user clicks "Diagnose" on an Argo CD application, the browser sends a POST request to the Go service, which queries the Kubernetes API for four signal types: Argo CD Application health and sync status, warning events scoped to the application's resources, pod container states with exit codes and restart counts, and the last 100 log lines from the first unhealthy pod. These signals are packed into one JSON payload and forwarded over HTTP to the Python agent on port 8081. On receiving the payload, the Python agent runs the heuristic router to select a specialist agent, injects FAISS-retrieved documentation into the agent's prompt context, and streams reasoning steps back through the Go service to the browser via Server-Sent Events (SSE). The final response is a JSON object containing the error summary, root cause, confidence score, and a copy able Git patch.

B. Heuristic A2A Routing

Running a classifier LLM on every incoming query would spend tokens just to pick the right agent, adding both latency and cost to a step that does not require natural language reasoning. The heuristic router avoids this by matching incoming signals against trigger lists defined in five agent cards:

Runtime Analyzer: OOM Killed, Crash Loop Back Off, Image Pull Back Off, exit codes, scheduling failures.

IRJCS: International Research Journal of Computer Science ISSN: 2393-9842
Volume 13, Issue 05, May 2026 https://www.irjcs.com/archives
https://doi.org/10.26562/irjcs.2025.v1305.04

Config Analyzer: Sync Error, Comparison Error, Out Of Sync, missing Config Map/Secret references.
Network Analyzer: Unhealthy events, connection refused DNS/TLS failures.
Storage Analyzer: Failed Mount, Provisioning Failed, PVC issues.
RBAC Analyzer: Forbidden, unauthorized, permission errors.

Matching runs in three passes: pod state reasons first (highest priority), then event reason fields, then keyword substring search across event messages. When multiple agents match, a fixed priority order (storage>rbac>network> config>runtime) resolves the tie. If nothing matches, the Runtime Analyzer handles the request as a fallback. Because the Kubernetes control plane generates strings like OOM Killed and Failed Mount deterministically, pattern matching against them is reliable without model inference. The ReAct pattern [16] and Tool former-style iterative tool calling [17] are applied within each specialist agent after routing is complete.

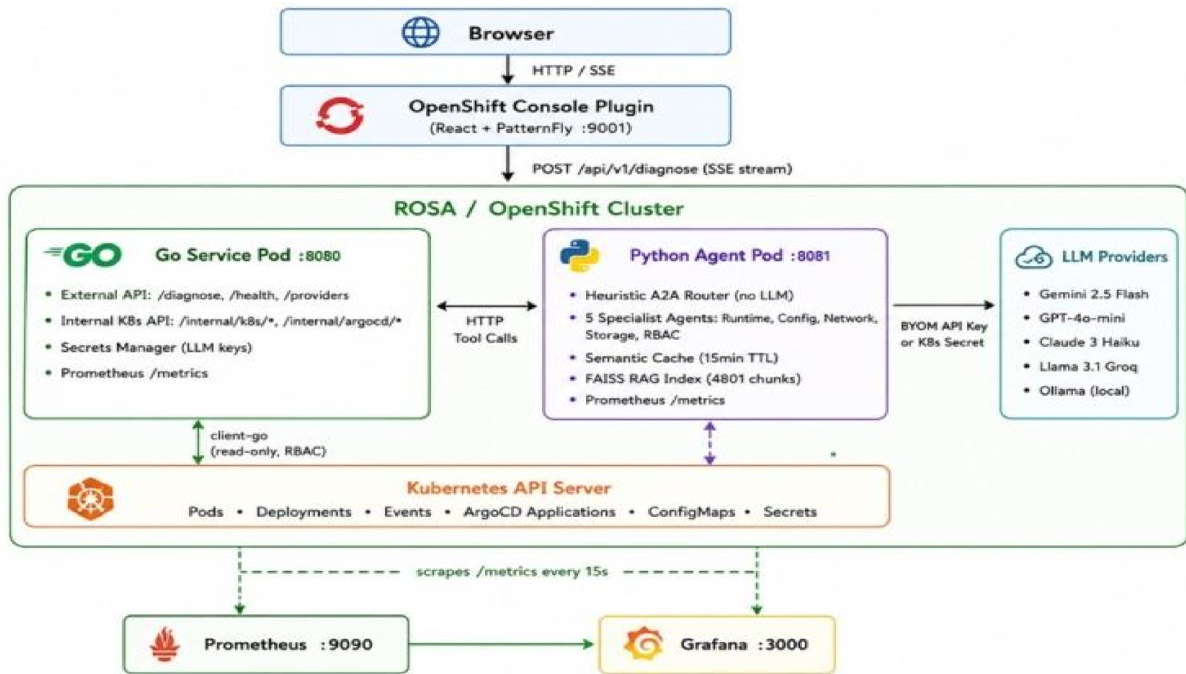


Fig.1. Argo AI System Architecture

C. Specialist Agent Design

Each agent needs a structured prompt to keep reasoning grounded and prevent hallucinated tool calls. All five agents are built on the Google ADK Llm Agent class with a system prompt containing four sections: (1) an explicit tool white list that blocks hallucinated tool names, (2) a context-first rule requiring the agent to analyze pre-loaded signals before calling extra tools,(3) a diagnostic decision tree specific to the agent's failure domain, and (4) anti-hallucination rules that demand evidence-backed conclusions. Seven diagnostic tools are exposed through Python Function Tool wrappers. When the agent invokes a tool (for example, get_pod_logs), the call travels from Python to the Go service via an internal HTTP POST endpoint, and the Go service executes the corresponding Kubernetes API query. Each tool response is truncated to 1,200 characters before entering the next reasoning round. A callback hook enforces a per-diagnosis budget of 3tool calls, hard-capped at 5, preventing runaway token consumption and keeping diagnosis cost predictable.

D. RAG Knowledge Base

LLMs answer Kubernetes questions more reliably when grounded in domain-specific documentation rather than parametric knowledge alone. The retrieval pipeline uses a pre-built FAISS inner-product index [18] containing 4,801 document chunks extracted from 809 files in the Open Shift Lights peed BYOK knowledge base [19]. Queries are encoded into 768- dimensional vectors using the all mpnet base v2 sentence transformer [20]. Chunks scoring above a similarity threshold of δ = 0.65 are injected into the agent's prompt context. Below this threshold, the system returns curated inline help covering 13 common Kubernetes error types.Fig.2shows this pipeline. By reusing the FAISS index that Open Shift Light speed maintains, Argo AI avoids building a separate document ingestion pipeline. The 0.65 threshold was chosen empirically to balance retrieval rate (81.0%) against accuracy (96.0%), as reported in Table IV.

E. Security Model

Prompt injection is the primary threat when an LLM sits between user input and a cluster API. Four concentric controls govern cluster access in Argo AI:

1) **Read-only operations**: Every Kubernetes API call is a GET request. No tool in the system issues create, update, or delete operations.
2) **Least-privilege RBAC**: The Go Service Account has read permissions limited to pods, events, config maps, and Argo CD Application objects.
3) **Secret isolation**: LLM API keys are stored in a Kubernetes Secret (Argo CD agent llm keys) accessible only to the Go pod. Keys are never written to logs.

4) **Zero K8s access for Python**: The Python pod has no Service Account binding. A successful prompt injection into the LLM cannot issue any Kubernetes API call because the pod itself has no credentials.

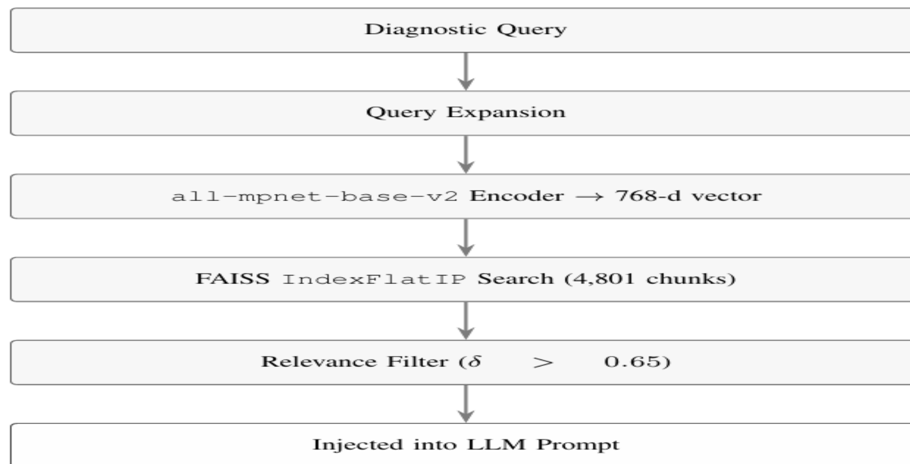


Fig.2. RAG retrieval pipeline

F. Caching and Log Pre-filtering

Repeated failures of the same type are common in Kubernetes: a pod stuck in Crash Loop Back Off generates identical events every few minutes, and diagnosing each occurrence from scratch wastes both tokens and wall-clock time. The diagnostic cache constructs a SHA-256 finger print from four fields:

$$f = \text{SHA256}\,(h_s \| s_s \| e_r \| p_r) \quad (1)$$

Where h_s is the health status, s_s is the sync status, e_r is the concatenated event reasons, and p_r is the concatenated pod state reasons. Timestamps and resource versions are excluded so that repeated failures of the same type produce a cache hit. Cache entries expire after 15 minutes for diagnosis results and 20 minutes for route decisions. Log pre-filtering runs in two stages. The Go service collects the final 100 tail lines of container logs, capped at 8KB. On the Python side, a regex filter keeps only lines containing diagnostic keywords (ERROR, FATAL, Exception, OOM Killed, failed). Lines carrying diagnostic signal are retained; heartbeat pings and health check output are discarded. In our test scenarios, this two-stage filter reduced the average token payload from 3,470 to 473 (Table III).

IV. EXPERIMENTAL EVALUATION

A. Setup

Testing ran on a local Mini kube cluster (v1.32.0) with Kubernetes v1.28 and Argo CD v2.10.4 deployed in the Argo CD name space. Both the Go and Python services ran as separate pods in the Argo CD agent name space. LLM calls went to hosted Gemini 2.5 Flash and Open AI GPT-4o-Mini end points, and to a local Ollama instance running Qwen 14B. Seven failure scenarios were injected by applying broken

Table- I Routing Accuracy across Seven Failure Scenarios

ID Failure Mode	Routed To	Result
TC-01 OOM Killed	Runtime Analyzer	Correct
TC-02 Image Pull Back Off	Runtime Analyzer	Correct
TC-03 Crash Loop Back Off	Runtime Analyzer	Correct
TC-04 Missing Config Map	Config Analyzer	Correct
TC-05 Out of Sync	Config Analyzer	Correct
TC-06 PVC Pending	Storage Analyzer	Correct
TC-07 Probe Failure	Network Analyzer	Correct

Table- II Diagnosis Latency: Cache Miss vs Cache Hit

ID	Failure	Miss (s)	Hit (ms)	Reduction
TC-01	OOM Killed	8.92	38	99.57%
TC-02	Image Pull	7.45	31	99.58%
TC-03	Crash Loop	9.15	41	99.55%
TC-04	Config Map	8.34	35	99.58%
TC-05	Out of Sync	6.80	29	99.57%
TC-06	PVC Pending	9.80	45	99.54%
TC-07	Probe Fail	8.75	39	99.55%
Avg		8.57	36.8	99.57%

A cache miss runs the full pipe line: signal ingestion, routing, FAISS retrieval, and LLM reasoning. On a hit, the stored result returns after a single SHA-256 comparison. Repeated synchronization failures for the same Argo CD application no longer trigger redundant LLM calls.

Table -III Log Token Payload Before and After Filtering

ID	Failure	Raw	Filtered	Savings
TC-01	OOM Killed	2,840	180	93.66%
TC-03	Crash Loop	4,120	750	81.79%
TC-07	Probe Fail	3,450	490	85.79%
Avg		3,470	473	87.08%

B. Routing Accuracy

The heuristic router dispatched all seven injected failures to the correct specialist agent, yielding100% accuracy with zero YAML manifests from the project's demo/ directory. Each scenario targets a different specialist agent and a different Kubernetes subsystem (runtime, configuration, storage, and network)

LLM tokens consumed (Table I).

Each scenario triggered exactly one agent's pattern list, and no tie-breaking was needed. TC-01 through TC-03 matched on pod state reasons (OOM Killed, Image Pull Back Off, Crash Loop Back Off), TC-04 and TC-05 matched on ArgoCD sync condition strings, TC-06 matched on the Failed Mount event reason, and TC-07 matched on the Unhealthy event reason.

C. Cache Latency

SHA-256 fingerprint caching reduced average diagnosis latency from 8.57s (full pipeline) to 36.8ms (cache hit), a 99.57% reduction (Table II).

D. Log Token Reduction

Regex-based log filtering cut the average token payload from 3,470 to 473, an 87.08% reduction across three log-heavy test cases (Table III). The filter retains lines matching diagnostic keywords (ERROR, FATAL, Exception, OOM Killed, failed) and discards heartbeat output and health check pings. API cost drops in proportion to the token reduction.

E. RAG Retrieval Accuracy

At the selected threshold of δ = 0.65, the RAG pipeline achieved 96.0% accuracy with an 81.0% retrieval rate across 100 diagnostic queries

Table IV- RAG Accuracy vs. Similarity Threshold δδ

Retrieval	Accuracy	Observation	
0.50	98.0%	62.0%	High recall, noisy context
0.60	89.0%	84.0%	Balanced, occasional noise
0.65	81.0%	96.0%	Selected default
0.75	54.0%	98.0%	Misses relevant chunks
0.85	12.0%	100.0%	Severe under-retrieval

Lower thresholds pull in more chunks but degrade accuracy: at δ = 0.50, retrieval reaches 98.0% but accuracy falls to 62.0% because noisy context misleads the LLM. Raising the threshold to δ =0.75 gains two percentage points of accuracy while cutting retrieval rate nearly in half, causing the system to miss documentation chunks that would ground the diagnosis. The 0.65 operating point gives agents enough relevant context without polluting the prompt window.

F. SSE Perceived Latency and UI Demonstration

The full cache-miss pipeline takes 8.57s on average, but the user sees activity within 320ms. An SSE stream delivers the first event (agent selection confirmation) to the browser inside that window. Tool call results and reasoning steps stream in as they happen, keeping the interface responsive throughout the diagnosis. Figs. 3–5 show three stages of a live diagnosis captured from the Argo AI console plug in running on an OKD cluster.

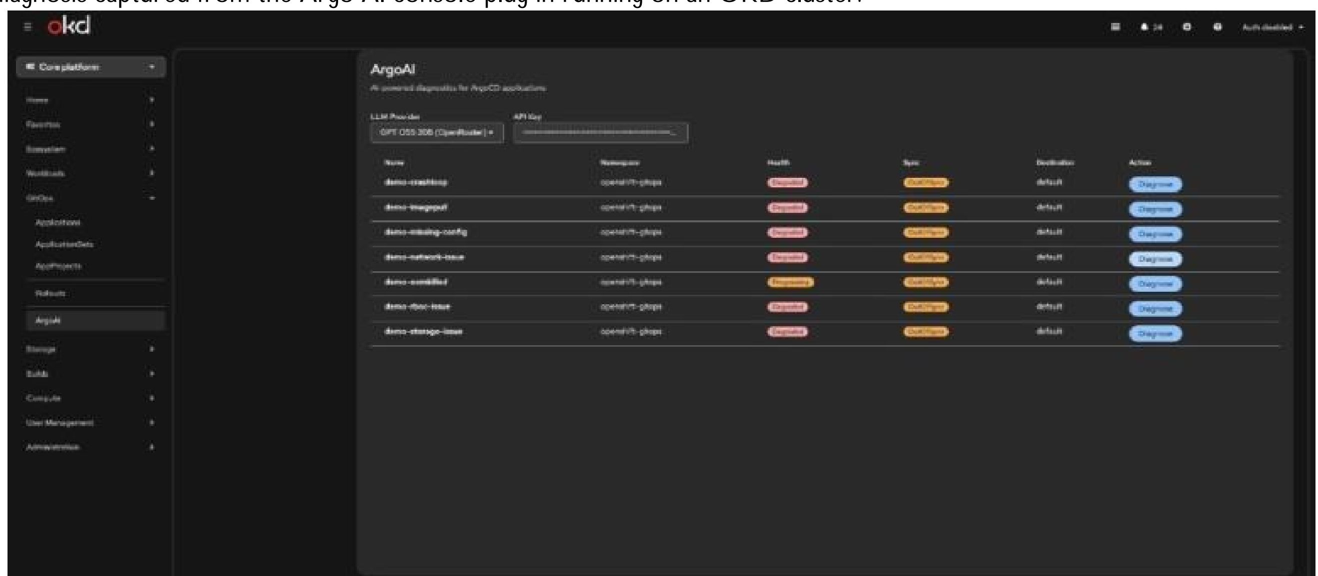


Fig. 3 - Application list with health /sync status.

IRJCS: International Research Journal of Computer Science ISSN: 2393-9842
Volume 13, Issue 05, May 2026 https://www.irjcs.com/archives
https://doi.org/10.26562/irjcs.2025.v1305.04

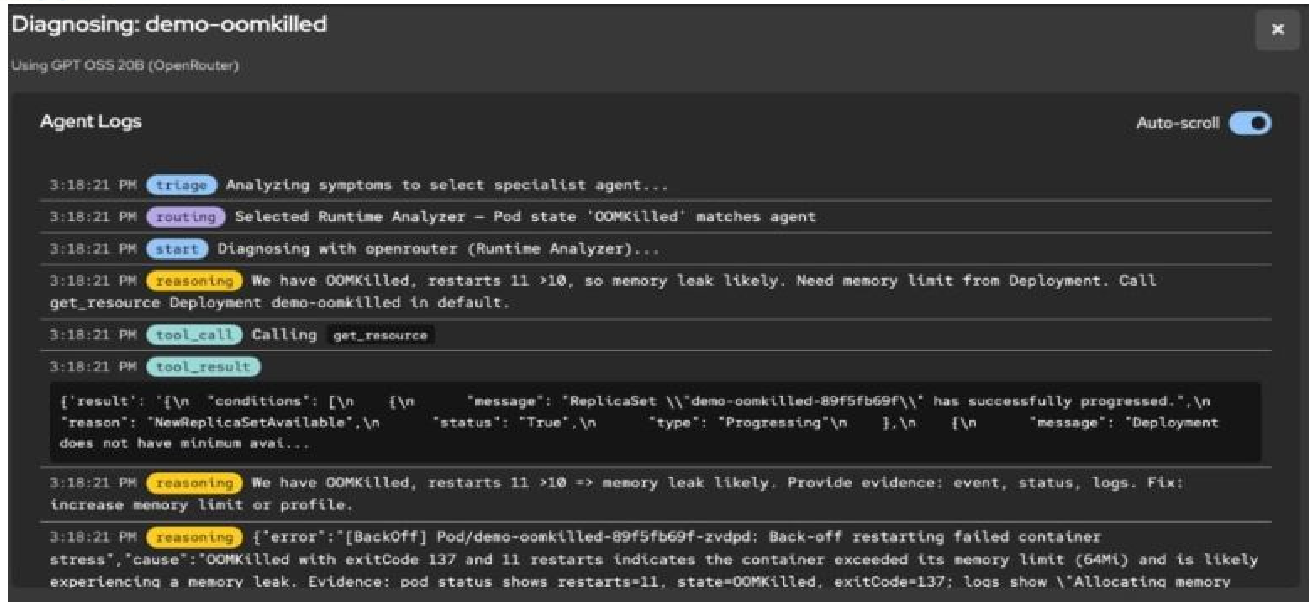


Fig.4. Live agent logs during OOM Killed diagnosis.

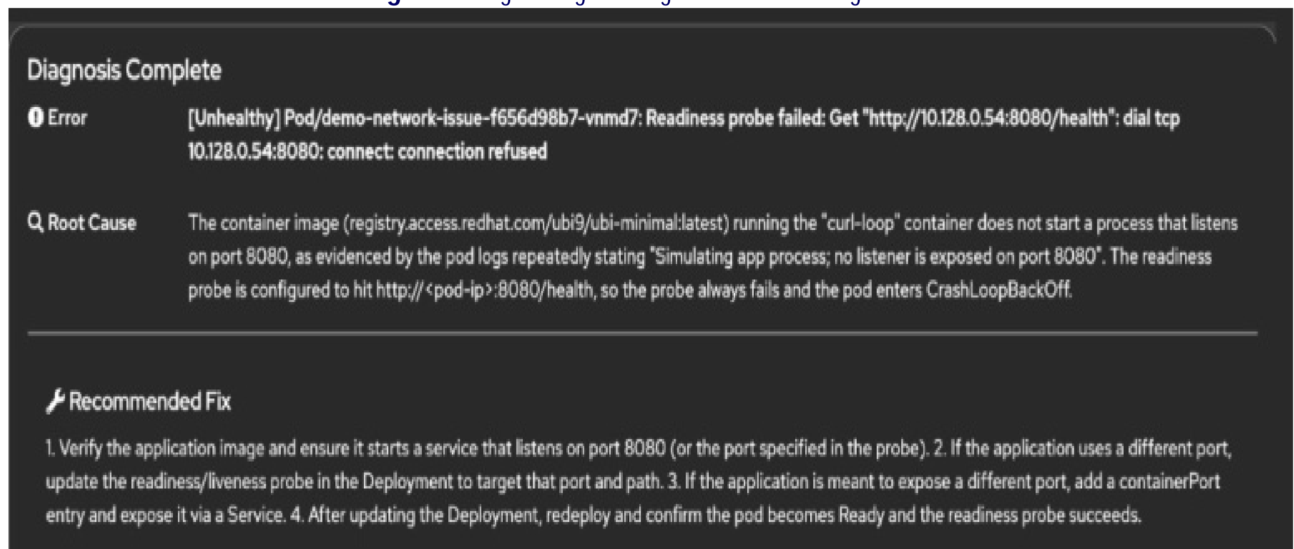


Fig.5. Diagnosis result for a probe failure.

DISCUSSION

The 100% routing accuracy in Table I stems from a specific property of Kubernetes: event reasons and pod state strings are not free text but enumerated codes generated by the kubelet and controller managers. Pattern matching against these codes is reliable as long as the trigger lists in each agent card cover the target failure modes. Composite failures expose the router's blind spot. If a pod fails for a reason that produces no recognized event string, or if two trigger patterns fire in a way that the fixed priority order mishandles, the system defaults to the Run time Analyzer. That fallback is conservative but not always precise. The cache fingerprint deliberately excludes timestamps and resource versions. For example, two OOM Killed events for the same application, occurring minutes apart, produce identical health status, sync status, event reasons, and pod state reasons. Including timestamps would defeat caching for the most common scenario: a pod failing repeatedly in the same way before anyone intervenes. The 87.08% token reduction in Table III was measured on containers that emit unstructured logs. Production workloads that use structured logging frameworks (Zap, Logback with JSON output) tag severity at the source, which means fewer noisy heartbeat lines to filter in the first place. The measured reduction may shift in either direction depending on the log format and the ratio of diagnostic lines to noise. Two scope limitations bound the current system. Fix quality depends on the underlying LLM: ArgoAI produces Git patch suggestions but cannot validate them against the repository's CI pipeline, so a patch that looks correct in the diagnosis context may filleting or integration tests. The Go collector also targets one ArgoCD name space per request, which means cross-namespace or multi-cluster failures require separate diagnosis runs.

CONCLUSION

Argo AI automates the failure triage that Argo CD engineers currently perform by hand, from signal collection and agent routing through to a structured root cause report. The heuristic router dispatched all seven test failures to the correct specialist agent while spending zero LLM tokens on routing, and SHA- 256 caching cut repeated diagnosis latency from 8.57 seconds to 36.8 milliseconds. All fixes that Argo AI suggests are YAML patches delivered as copyable Git commands.

No cluster resources are created, modified, or deleted during diagnosis. Five extensions are planned for production readiness: (1) automatic Git pull request generation from recommended patches, (2) hybrid retrieval combining dense FAISS vectors with BM25 keyword search, (3) Alert manager web hook integration for triggering diagnosis on alert firing, (4) Kubernetes CRDs for declarative diagnostic policy management, and (5) multi-cluster telemetry aggregation across federated ArgoCD deployments.

ACKNOWLEDGMENT

The authors thank the Git Op steam at RedHat for providing access to the Open Shift Light speed BYOK knowledge base and for technical guidance during the development of Argo AI.

AUTHOR CONTRIBUTION STATEMENT

Conceptualization: Tejas Soham, Rekha B.S
Literature Review and Methodology design: Tejas Soham
Software: Rekha B.S
Validation: Tejas Soham
Formal Analysis: Tejas Soham
Investigation: Rekha B.S
Resources: Tejas Soham
Data Curation: Tejas Soham
Writing original draft preparation: Rekha B.S
Writing review and Editing: Tejas Soham
Visualization: Tejas Soham
Supervision: Tejas Soham
Project Administration: All authors have read and agreed to the published version of the manuscript
Conflict of interest: The authors declare no conflicts of interest.
Data availability statement: Data supporting these findings are available within the article, at https://doi.org/10.26562/irjcs.2026.v1305.04, or upon request.

Publisher's note
AM Publications, India. IRJCS (International Research Journal of Computer Science) Journals stays neutral with regard to jurisdictional claims in published maps and institutional affiliations. All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher. https://ampublications-india.com/

REFERENCES

1. B.Burns, J.Beda, and K.Hightower, Kubernetes: Up and Running,2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2019, ISBN: 978-1-492-04653-0.
2. A.Richardson,"Operations by pull request: The GitOps approach,"Weave works, White Paper, 2017. [Online]. Available: https://www.weave.works/technologies/gitops/
3. Argo Project Contributors, "Argo CD: Declarative GitOps continuous delivery for Kubernetes, "Cloud Native Computing Foundation (CNCF),2023. [Online]. Available: https://argo-cd.readthedocs.io
4. Red Hat, Inc., "Open Shift Light speed service documentation," 2024.[Online]. Available: https://docs.openshift.com/container-platform/latest/lightspeed/
5. Argo proj Labs Contributors, "Assistant for ArgoCD," GitHub repos-itory, 2025. [Online]. Available: https://github.com/argoproj-labs/assistant-for-argocd
6. Red Hat Emerging Technologies, "OpenC law: Agentic infrastructure diagnostics for Kubernetes and Open Shift," GitHub repository, 2025.[Online]. Available: https://github.com/redhat-et/openclaw-infra
7. M.Seyedkazemi Ardebili and A.Bartolini, "Kube Intellect: A modular LLM-orchestrated agent frame work for end-to-end Kubernetes management, " arXiv preprint, arXiv:2509.02449, Sep. 2025.
8. Y.Chen, J.Pan, J.Clark, Y.Su,N.Zheutlin, B.Bhavya,R.R.Arora, Y. Deng, S. Jha, and T. Xu, "STRATUS: A multi-agent system for autonomous reliability engineering of modern clouds," in Proc. Adv.Neural Inf. Process. Syst. (NeurIPS), Vancouver, BC, Canada,Dec.2025,pp.1–12.
9. X.Zhang,Q.Wang,M. Li,Y.Yuan,M.Xiao,F.Zhuang, and D.Yu,"TAMO: Fine-grained root cause analysis via tool-assisted LLM agent with multi-modality observation data in cloud-native systems," arXiv preprint, arXiv:2504.20462, Apr. 2025.
10. Google, "Agent Development Kit (ADK): Build, deploy, and orchestrate AI agents, "2025. [Online]. Available: https://google.github.io/adk-docs/
11. Berri AI, "LiteLLM: Call100+LLM APIs using the Open AI format," GitHub repository, 2024. [Online]. Available: https://github.com/BerriAI/litellm
12. P.Notaro, J.Cardoso, and M.Gerndt, "A systematic mapping study in AIOps," in Proc. Int. Conf. Service-Oriented Comput. Workshops (ICSOC), Dubai, UAE, Dec. 2020, pp. 110–123, https://doi.org/10.1007/978-3-030-76352-715
13. Q. Lin et al., "Predicting node failure in cloud service systems," in Proc. ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw.Eng. (ESEC/FSE), Lake BuenaVista, FL, USA, Nov.2018, pp. 480–490, https://doi.org/10.1145/3236024.3236060.

14. Y.Chen et al.,"Automatic root cause analysis via large language models for cloud incidents," in Proc.Eur.Conf. Comput.Syst. (EuroSys), Athens, Greece, Apr. 2024, pp. 674–688, https://doi.org/10.1145/3627703.3629553.
15. M.Barletta, M.Cinque, C.DiMartino, Z.T.Kalbarczyk, and R.K.Iyer,"Mutiny! How does Kubernetes fail, and what can we do about it?," in Proc. 54th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN), Brisbane, QLD, Australia, Jun. 2024, pp. 1–14, https://doi.org/10.1109/DSN58291.2024.00016.
16. S.Yao et al., "ReAct: Synergizing reasoning and acting in language models," in Proc. Int. Conf. Learn. Representations (ICLR), Kigali,Rwanda, May 2023, pp. 1–20.
17. T.Schick et al., "Tool former: Language models can teach themselves to use tools," in Proc. Adv. Neural Inf. Process. Syst. (NeurIPS), NewOrleans, LA, USA, Dec. 2023, pp. 1–15.
18. J.Johnson, M.Douze, and H.Je´gou, "Billion-scale similarity searchwith GPUs," IEEE Trans. Big Data, vol. 7, no. 3, pp. 535–547, Jul.2021, https://doi.org/10.1109/TBDATA.2019.2921572.
19. RedHat,Inc.,"Open Shift Light speed BYOK knowledge base, "Quay.io container image,2024. [Online]. Available: https://quay.io/devtoolsgitops/argocdlightspeedbyok
20. N.Reimers and I.Gurevych, "Sentence-BERT: Sentence embeddings using Siamese BERT-networks," in Proc. Conf. Empirical Methods Natural Lang. Process. Int. Joint Conf. Natural Lang. Process.(EMNLP-IJCNLP), HongKong, China, Nov.2019, pp.3982–3992,doi:10.18653/v1/D19-1410.
21. P.Lewis et al., "Retrieval-augmented generation for knowledge-intensive NLP tasks," in Proc. Adv. Neural Inf. Process. Syst. (NeurIPS), Virtual,Dec. 2020, pp. 9459–9474.
22. A.Vaswanietal.,"Attention is all you need," inProc. Adv.Neural Inf. Process. Syst. (NeurIPS), Long Beach, CA, USA, Dec. 2017, pp.5998–6008.
23. S.Ram´ırez,"Fast API: Modern, fast(high-performance) web frame work for building APIs with Python," 2023. [Online]. Available: https://fastapi.tiangolo.com
24. B.Beyer, C.Jones, J.Petoff, and N.R.Murphy, Site Reliability Engineering: How Google Runs Production Systems. Sebastopol, CA,USA: O'Reilly Media, 2016, ISBN: 978-1-491-92912-4.
25. D.Merkel, "Docker: Lightweight Linux containers for consistent development and deployment," Linux J., vol. 2014, no. 239, Art. no. 2, Mar.2014.