Agent-Based SRE: Automated Diagnosis and Mitigation in K8s

1st Marcello Martini
Politecnico di Milano
marcello.martini@mail.polimi.it

***Abstract*—Managing modern cloud-native infrastructures requires effective, automated solutions for timely incident detection and remediation. This paper presents a proof-of-concept AI agent, built using the LangGraph framework on top of LangChain, designed to autonomously manage and respond to faults in Kubernetes-based microservices. Leveraging integrations with observability tools and databases through the Model Context Protocol (MCP), the agent simplifies resource connectivity and enables advanced workflow orchestration. The proof-of-concept implementation incorporates optimization techniques to enhance diagnostic efficiency and reduce operational costs. Evaluation in a benchmark environment demonstrates the practical viability of the approach, highlighting the potential of LLM-powered agents for Site Reliability Engineering tasks in complex cloud-native systems.**

***Index Terms*—Agentic AI, Site Reliability Engineering (SRE), Kubernetes, Large Language Models (LLMs), LangGraph, Model Context Protocol (MCP), Microservices, Automated Diagnosis, Incident Mitigation, Cloud-native Infrastructure**

Codebase: github.com/martinimarcello00/SRE-agent

I. Introduction

Modern cloud-native infrastructures (e.g., Kubernetes) are complex and prone to faults. Site Reliability Engineers (SREs) face challenges in rapidly diagnosing and mitigating incidents in distributed systems. As organizations increasingly adopt microservices architectures, the operational overhead and cognitive load on SRE teams have grown significantly. Effective management of these dynamic environments requires timely detection, root cause analysis, and remediation of failures to ensure high availability and performance.

This project presents a proof-of-concept AI agent developed using the LangGraph framework, leveraging large language models (LLMs) to autonomously diagnose and respond to incidents within a Kubernetes cluster running a benchmark microservice architecture. The agent integrates with observability tools (Prometheus), persistent vector databases (ChromaDB), and the Kubernetes API through the standardized Model Context Protocol (MCP), simplifying resource connectivity and enabling sophisticated workflow orchestration. The implementation focuses on demonstrating feasibility and core functionality, incorporating optimization techniques to enhance diagnostic efficiency and cost-effectiveness, thus laying the foundation for further development of autonomous SRE agents.

The work illustrates the potential of AI-driven agents to augment traditional SRE workflows, improve system reliability, and minimize downtime in complex cloud-native environments.

II. Related work

Agentic AI, an emerging paradigm in artificial intelligence, refers to autonomous systems designed to pursue complex goals with minimal human intervention. Unlike traditional AI, which depends on structured instructions, Agentic AI demonstrates adaptability and advanced decision-making, enabling it to operate dynamically in evolving environments. A key application of this paradigm is in **Artificial Intelligence for IT Operations** (**AIOps**), which leverages AI to automate and streamline IT operations. By applying agentic principles to AIOps, systems can gain a **holistic view** of an entire infrastructure, making tasks such as root cause analysis and issue resolution faster and more effective.

As the complexity of cloud environments grows, so does the need for sophisticated AI agents to manage them. This has led to significant research from major technology companies. Chen et al. [1] from Microsoft presented **AIOpsLab**, a benchmark suite for evaluating AI agents in autonomous cloud management. This framework is designed to deploy microservice environments, inject diverse faults, generate realistic workloads, and provide an Agent-Cloud Interface (ACI) to facilitate the end-to-end evaluation of AI agents. Similarly, Jha et al. [2] from IBM introduced **ITBench**, a framework with a collection of benchmarks for assessing AI agents in real-world IT automation tasks, including Site Reliability Engineering (SRE). The development of these comprehensive evaluation frameworks highlights the increasing demand for robust methods to assess agent performance in realistic, interactive settings.

To construct agents capable of tackling these complex tasks, developers are turning to specialized frameworks. **LangChain**[1] offers a standard interface for integrating Large Language Models (LLMs) with various tools and databases, simplifying development. Building on this, **LangGraph**[2] provides a more advanced framework for creating resilient and stateful agents by representing workflows as graphs. This structure is particularly well-suited for SRE tasks, which often involve complex, multi-step processes with loops and conditional branching. LangGraph's architecture allows for the

[1]www.langchain.com/langchain
[2]www.langchain.com/langgraph

development of sophisticated and flexible agent behaviors that can be monitored and easily debugged using the **LangSmith**[3] observability platform, which enables to debug, test, and monitor AI app performance.

A primary challenge in developing production-ready AI agents is the integration of external tools and data sources. Historically, this required custom ad hoc connectors for each resource. To address this, the proposed project adopts the **Model Context Protocol**[4] (**MCP**), an open standard introduced by Anthropic to streamline how applications connect tools and provide context to LLMs. MCP acts as a universal interface, abstracting away the complexity of individual integrations and promoting a standardized, state-of-the-art approach to building and scaling agent capabilities.

Recent advancements in **retrieval-augmented generation** (**RAG**) have shown the effectiveness of integrating pretrained language models with external knowledge retrieval through vector databases (Lewis et al. [3]). Vector stores enable efficient similarity search over high-dimensional embeddings, facilitating rapid access to relevant historical information. Persistent vector databases such as **ChromaDB**[5] provide optimized storage, indexing, and querying of embeddings alongside rich metadata, enhancing AI agents' reasoning capabilities by enabling contextual reference to prior incident reports and operational knowledge.

The present work contributes a **proof-of-concept AI agent** for autonomous fault diagnosis and mitigation in Kubernetes-based microservices. Key contributions include leveraging the **Model Context Protocol** (**MCP**) for standardized and simplified integration with observability tools and APIs, introducing **ChromaDB** as a persistent vector database for efficient incident history retrieval and similarity-based reasoning, and developing a resilient agent using the **LangGraph** framework. Additionally, this work emphasizes optimization techniques aimed at **reducing token usage** during LLM interactions, thereby improving efficiency and lowering operational costs. These elements collectively demonstrate the practical feasibility and performance benefits of **AI-powered Site Reliability Engineering** in dynamic cloud-native environments.

III. Methods

This section explains how the SRE Agent works, focusing on its main operating method and improvements made to increase efficiency and troubleshooting. The main metric used for improvement is the number of tokens the model consumes to finish a task, since this directly affects the cost when using the **OpenAI public API** (**gpt-5-mini**[6]).

A. Agent Pattern: ReAct

Yao et al. [4] presented the **Reasoning and Acting** (**ReAct**) pattern, an approach that uses LLMs to generate both reasoning traces and trask-specific action in an interleaved manner, improving the decision-making and problem-solving capabilities of LLMs. The framework relies on several core components: the **language model** itself, a **prompting strategy** to structure the alternation between reasoning and actions, and **mechanisms for interfacing with external tools** or environments to execute decisions and incorporate feedback. The ReAct pattern is employed to build the SRE Agent since it provides a way to explore and take decision about the current Kubernetes status: at each iteration, the agent can call one of the tools to interact either with Kubernetes or Prometheus, send a request specifying parameters etc. (e.g. making a tool call) which will be sent to a *tool* node, a LangGraph node with the task of routing the request to the proper tool, waiting the response, and forwarding it to the LLM node. The LLM will analyse the history of tool calls/response and decides the next action to take (wether to call another tool or submit the solution). This approach suits well the SRE diagnosis phase in which the goal is to identify the Root Cause of the problem by looking at all the metrics/logs from the entire Kubernetes Cluster.

Another pattern that could have been used is **Plan-and-Execute**, which follows a "plan first, execute later" strategy, dividing tasks into two phases. First, the system analyzes objectives, breaks them into sub-tasks, and creates an execution plan; then, it sequentially carries out each sub-task, processes results, and adjusts the plan as needed. This approach improves task organization and adaptability.

For the SRE agent, the ReAct pattern was chosen instead, as cluster analysis can vary with each instance and there is no true "standard plan" to follow. In this context, generating detailed execution plans would be inefficient, since they would likely change completely after just two or three iterations since the analysis flow depends largely on information gathered from tools at each previous step.

The simplest implementation of the ReAct pattern (figure 1) using LangGraph consists of two nodes that loop until the LLM node determines that the analysis is complete. At that point, it outputs the root cause of the problem and exits the loop by transitioning to the end node. The **graph state**, which refers to the information shared among all nodes, is composed solely of the message history between the LLM and tool nodes, with new messages appended at each iteration.

This approach is effective with respect to finding the root cause of the incident but lacks optimization. Since the entire message history is appended to the prompt at each iteration, the agent wastes a lot of tokens because it does not employ a properly structured schema that would allow information to be stored in a condensed and optimized way. This problem affects both the cost per task completion and the time required, as the prompt input, and thus the execution time, increases with the number of iterations.

B. Optimizing the Agent: Context Window Management

To optimize the agent, **context window management** is applied by limiting the amount of interaction history passed to the model at each step. Context window management refers

[3]www.langchain.com/langsmith
[4]modelcontextprotocol.io
[5]www.trychroma.com
[6]platform.openai.com/docs/models/gpt-5-mini

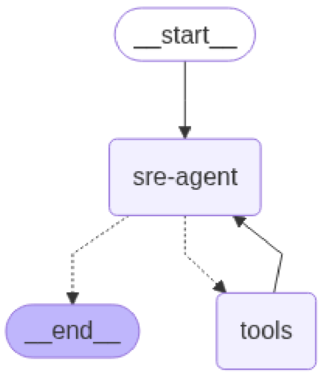

Fig. 1. ReAct simplest pattern: the agent is composed by two nodes: the LLM node (*sre-agent*) which analyse the output of the *tools* and decide the next action and the tool node, which routes the tool calls to the tools.

to the process of selecting which parts of the conversational or task history are included as input for a large language model (LLM), given that LLMs can only process a finite amount of text at once. In this approach, only the last seven iterations between the tool and LLM nodes are retained, rather than the full message history. This reduces the number of tokens required as input, improving efficiency and lowering costs. However, this optimization may sometimes exclude important earlier information. For instance, in root cause analysis tasks, the agent might need access to earlier logs or messages to accurately diagnose an issue. Discarding older messages can result in the agent repeating actions or tool calls (such as multiple `kubectl_get` requests) when information is missing, ultimately increasing both the number of iterations and total tokens used, and potentially slowing down task completion.

C. Optimizing the Agent: Structured Graph Schema

LangGraph defines a **graph schema** that represents the entire graph execution across the nodes. Each node can read from and append to or update the contents of the graph. The default (and simplest) schema consists only of the chat history, but as discussed in subsection III-A, this approach is not optimized. To reduce token usage and thus optimize the agent, a **structured graph schema** is employed. A structured schema is a **typed dictionary** that contains organized content and allows only relevant information gathered at each step of the analysis to be stored, avoiding the need to send the entire message history every time.

For the SRE agent, the state comprises three main components:

- *Insights*: a list of strings updated at each iteration with observations derived from tool responses (for example, after a `kubectl_log` call that returns recent pod logs, an insight might be an error message or a critical metric detected in the logs);
- *Previous steps*: a record of all tool calls performed by the LLM node;

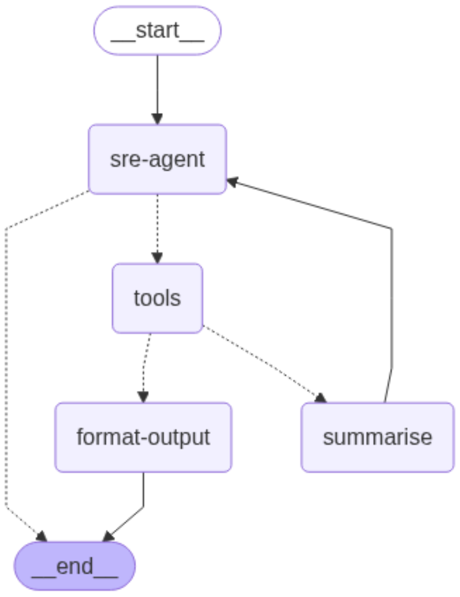

Fig. 2. ReAct pattern with structured graph schema: two more nodes are added, the *summarise* node which extract the key insights and the *format-output* node which formats the the final response after the agent finishes the analysis.

- *Response*: a structured output generated when the analysis is complete, containing both a description of the incident and the reasoning process that led to its identification.

To implement this pattern, an additional node, the *summarise* node, is introduced into the graph. This node takes as input the current insights, previous steps, and the last two messages (i.e., the tool call and its response). It then updates the graph state by appending a new insight and recording the most recent step taken. Connected to the *tool* node, as illustrated in figure 2), the *summarise* node fits into the workflow as follows: at each iteration, the *LLM* node generates a tool call based on existing insights and steps, the *tool* node executes the request, and finally, the *summarise* node updates the graph state schema.

To ensure that the summarise node produces output in the correct format, **LangChain's structured output feature** is used. This allows you to define a typed Pydantic class that specifies the exact structure and fields expected in the output. The LLM is then instructed to return its response in a JSON format matching this schema.

The *format-output* node is an additional node which takes the formatted response submitted by the agent and format it in a proper output for improving the readability.

By using this approach, the *LLM nodes* (the *SRE Agent*, which decides the next action, and the *summarise* node, which extracts key insights) never see the full message history, but only the relevant information. For example, the summarise prompt is generated by providing the previous insights, steps taken, and the last two messages. This dramatically **reduces token usage** and thus **lowers the cost** per task.

Summarise node prompt

You are an autonomous SRE agent for Kubernetes incident diagnosis.
Context:
Previous Insights:
{insights}
Previous Steps:
{prev_steps}
Below are the latest two messages:
{last_two_messages}
Instructions:

1) From the latest two messages, extract the most important new insight relevant for incident diagnosis or mitigation. Summarize it concisely.
2) Write a concise description of only the most recent action taken, including the tool used (not the whole list).

D. Mitigation plan generation

After developing the diagnostic component, the next task for an SRE agent is to **generate or execute a mitigation plan**. For the scope of this project (whose goal is to demonstrate that an LLM-based agent can detect incidents within a microservice network) the agent is limited to producing a mitigation plan without executing it, although with minor modifications, the agent could be enabled to take action within the Kubernetes cluster.

The mitigation plan generator is implemented using a ReAct pattern agent, which has access to tools exposed via MCP servers to Kubernetes. This allows the agent to check whether the proposed mitigation plan is feasible and to gather any additional details required to support its recommendations. Additionally, it utilizes **ChromaDB**, a vector database, to store and retrieve previous similar incident reports. In this implementation, ChromaDB is configured with **persistent storage**, ensuring that incident reports and their embeddings are retained across sessions and restarts.

The agent returns a structured output containing three fields:

- *Mitigation steps*: a list of actions to be executed in order to resolve the current incident.
- *Mitigation plan overview*: a concise summary of the mitigation strategy.
- *Is previous incident*: a boolean indicating whether the proposed plan was found among previously recorded incidents (i.e., if the incident has occurred before).

These fields are incorporated into the structured graph schema so that the generated recommendations are shared across all nodes for subsequent instructions. The *format-output* node is redesigned to produce the incident report in Markdown format. Furthermore, after generating the Markdown report, a *conditional node* checks whether the incident is new (i.e., the query to the database did not find any similar incident report). If it is a new incident, the report is stored and indexed in the vector database.

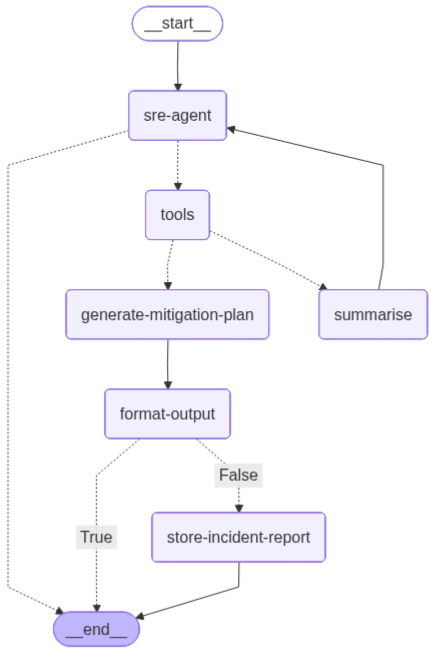

Fig. 3. SRE agent graph with mitigation plan generator: the agent is composed by two main components: the *sre-agent* which is in charge of finding the diagnosis and the *generate-mitigation-plan* agent in charge of generating the mitigation plan given the diagnosis.

The final structure of the SRE Agent is illustrated in figure 3.

E. RAG-based incident history lookup

In the latest version, the agent first runs a quick, low-cost pass to **collect early symptoms**. These symptoms are stored in the graph state and used to build one **semantic search query** against a ChromaDB vector store of past incident reports. If a close match is found, the agent checks that the symptoms and main signals are the same; when confirmed, it reuses the old diagnosis, avoiding extra tool calls and saving time and tokens. If the match is weak or no result is found, the agent continues the normal ReAct analysis. After finishing, it saves the new report (with symptoms) to ChromaDB for future lookups. In addition to incident reports, the retrieval layer can also incorporate company-specific **Standard Operating Procedures** (SOPs) and **runbooks**, allowing the agent not only to learn from historical cases but also to align with established best practices and resolution guidelines defined by the organization.

F. MCP Servers

For the connection of the tools, **MCP servers** are used as they represent the current standard protocol for integrating external services with language model agents. Three MCP servers are configured: **Kubernetes**[7] (providing access to

[7]github.com/Flux159/mcp-server-kubernetes

Kubectl), **Prometheus**[8], and **ChromaDB**[9]. Stdio communication is chosen instead of HTTP, as it simplifies local development by avoiding port management and reducing configuration overhead. LangGraph acts as an **MCP client**, connecting to multiple MCP servers simultaneously to orchestrate tool usage. For Kubernetes, all operations that could modify kubeconfig or cluster state are disabled, ensuring the agent has access only to **read-only commands** for safe experimentation.

IV. Experimental Setup

This section explains how the experiments were set up. It covers the test environment, how the agent was run, how its activity was monitored, and how its performance was evaluated.

A. Test Environment

All experiments were conducted on a MacBook Pro M1 with 16 GB of RAM. The Kubernetes cluster was created using **Kind** (**Kubernetes in Docker**) with a custom configuration to expose the Prometheus server port, making it accessible to the MCP server.

The testbed was set up using **AIOpsLab**, a Microsoft framework for evaluating AIOps agents. In this work, AIOpsLab was used solely to provision the Kubernetes environment, inject faults, and generate workload traffic; it was not used for controlling interactions between the agent and the cluster or for evaluation, as the MCP protocol was adopted to expose and coordinate services for the agent. AIOpsLab includes support for deploying several microservice applications from the **DeathStarBench** [5] benchmark suite. Faults such as misconfigurations or network delays are injected using **Chaos Mesh**[10], while workload generation is performed with **wrk2**[11].

To set up the experiments, the Kind cluster is first created. Then, using the `cli.py` client from AIOpsLab, one of the available experiments can be configured and launched. Due to hardware limitations, only the `hotel-reservation` network could be run, and only with limited levels of injected network delay. Occasionally, these constraints led to issues with the Prometheus server during metric collection.

B. Agent Deployment and Execution

The agent was developed and executed within a Jupyter Notebook environment, allowing outputs (including intermediate results and incident reports) to be stored within the notebook file itself. A custom function was implemented to save incident reports in Markdown format for documentation and further analysis. Additionally, a dedicated tool was created to record the distribution of tool calls made by the agent during execution. The agents were also designed to be compatible with **LangGraph Studio**[12], an integrated development environment (IDE) provided by LangChain, which facilitates interactive debugging and development of agent workflows.

[8]github.com/idanfishman/prometheus-mcp
[9]github.com/chroma-core/chroma-mcp
[10]chaos-mesh.org
[11]github.com/giltene/wrk2
[12]docs.langchain.com/langgraph-platform/langgraph-studio

C. Monitoring Framework

Multiple mechanisms are employed to monitor and analyze the agent's execution. First, the graph state maintains a record of all messages exchanged by the LLM and tool nodes during execution, allowing the entire history of interactions to be reviewed for correctness checks and further analysis. Additionally, the agent stores each step performed, along with its reasoning process, to facilitate debugging and improve explainability.

Finally, the entire graph execution is traced and monitored using the **LangSmith platform**. LangSmith enables detailed inspection of the workflow by tracking token usage across all nodes in the graph, recording tool call statistics, and providing access to all intermediate graph states. This monitoring framework ensures transparency, enables performance analysis, and supports in-depth debugging throughout the agent's operation.

D. Evaluation Methodology

Due to hardware limitations, it was challenging to comprehensively assess the agent's performance. Only a single application could be executed under these constraints, and Prometheus metrics were sometimes unreliable because the server occasionally failed to collect data. The issue `hotel reservation misconfiguration geo-pod` was considered, in which a faulty pod image was injected, causing the container to attempt to connect to the wrong port of its MongoDB and subsequently crash. Despite these issues, the primary metrics considered were **total token usage** and graph **execution time**. While execution time is not entirely reliable given the hardware constraints, it still provides a useful point of comparison since all tests were run on the same testbed. Additionally, manual inspection of the incident reports was performed to verify whether the agent identified the correct solution.

V. Results

As shown in Table I and Fig. 4, when only the diagnostic agent is considered (without mitigation), the **structured graph schema reduces token usage** by sending a **distilled context** (only the executed steps and current insights) instead of the full chat history. This saving **introduces a small delay** in execution time compared to the plain ReAct baseline, because the graph adds nodes and **requires extra LLM calls** to analyze and extract insights. A **reduced context window brings limited benefit** and can backfire: details older than about seven messages are dropped, leading to repeated tool calls and extra tokens.

For **mitigation plans**, the version without incident fetching is faster and cheaper because it skips symptom discovery and semantic retrieval. When incident fetching is enabled, performance improves after the first occurrence: later runs take about **half the time** and use around **36k fewer tokens**. This improvement is due to the **retrieval layer acting as a cache**: the first occurrence pays the cost to build a compact incident record, while later runs retrieve and adapt it, avoiding most exploratory queries and long reasoning chains.

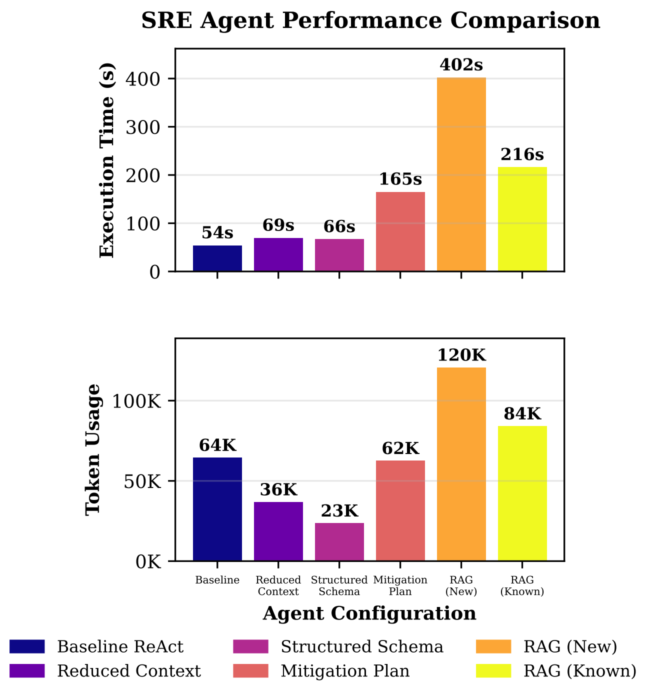

Fig. 4. Performance of the SRE agent across configurations. The structured schema reduces tokens with a small latency increase versus Baseline ReAct (54s→66s; 64K→23K tokens). For mitigation, incident fetching (RAG) is costly on first occurrence (402s, 120K) but amortizes on repeats (216s, 84K; ∼2× faster, -36K tokens).

Overall, **MCP connectors increase generalization** and ease integration across resources, but their verbose tool calls **add token overhead**; in contrast, the custom AIOpsLab setup is more token-efficient but harder to extend.

Throughout the experiments, the **agent consistently discovered the root cause of the injected faults**. In some cases, it found faults associated with the testbed environment itself, in particularly the noticeable delays in the Consul controller due to limited hardware resources. These operational issues caused the agent to detect real failures beyond the injected anomalies.

TABLE I
EXECUTION TIME AND TOKEN USAGE ACROSS AGENT CONFIGURATIONS

Agent configuration	Execution time (s)	Token usage
Baseline ReAct Agent	53.70	64542
Reduced Context Agent	69.25	36869
Structured Schema Agent	**66.42**	**23644**
Mitigation Plan Agent	164.81	62595
RAG (new incident)	401.73	120741
RAG (already happened)	215.81	84177

VI. Limitations and Future Work

This project shows that Agentic AI can support SRE tasks. Due to hardware limits, the agent was tested on a single microservices environment and on one fault type only, because the Prometheus server did not provide reliable metrics. Future work will test the agent configurations across the full AIOpsLab test suite to provide a more complete view of performance. Additional MCP connectors will be explored, such as Grafana Loki[13] for log collection and tools like Firecrawl[14] to retrieve online service documentation. A **human-in-the-loop step** may be added to the mitigation phase so the agent can execute the plan with explicit human approval and feedback. Finally, because the agent is implemented with LangGraph, different language models can be plugged in, including **custom Small Language Models** (Belcak et al. [6]) fine-tuned for specific diagnostic tasks, for example, a model trained to detect faults from pod logs.

VII. Conclusion

This project demonstrates the potential of agentic AI, leveraging the LangGraph framework and the Model Context Protocol, to effectively **support Site Reliability Engineering tasks** in complex Kubernetes environments. By implementing an optimized ReAct pattern with a structured graph schema, the agent achieves **significant reductions in token usage** while maintaining reliable incident diagnosis capabilities. The integration of a mitigation plan generator and incident history retrieval further enhances operational efficiency and reuse of past knowledge. Despite hardware and data collection limitations, the promising results highlight the viability of AI-driven autonomous agents for cloud-native system management.

References

[1] 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," Jan. 2025, arXiv:2501.06706 [cs]. [Online]. Available: http://arxiv.org/abs/2501.06706

[2] S. Jha, R. Arora, Y. Watanabe, T. Yanagawa, Y. Chen, J. Clark, B. Bhavya, M. Verma, H. Kumar, H. Kitahara, N. Zheutlin, S. Takano, D. Pathak, F. George, X. Wu, B. O. Turkkan, G. Vanloo, M. Nidd, T. Dai, O. Chatterjee, P. Gupta, S. Samanta, P. Aggarwal, R. Lee, P. Murali, J.-w. Ahn, D. Kar, A. Rahane, C. Fonseca, A. Paradkar, Y. Deng, P. Moogi, P. Mohapatra, N. Abe, C. Narayanaswami, T. Xu, L. R. Varshney, R. Mahindru, A. Sailer, L. Shwartz, D. Sow, N. C. M. Fuller, and R. Puri, "ITBench: Evaluating AI Agents across Diverse Real-World IT Automation Tasks," Feb. 2025, arXiv:2502.05352 [cs]. [Online]. Available: http://arxiv.org/abs/2502.05352

[3] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," Apr. 2021, arXiv:2005.11401 [cs]. [Online]. Available: http://arxiv.org/abs/2005.11401

[4] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "ReAct: Synergizing Reasoning and Acting in Language Models," Mar. 2023, arXiv:2210.03629 [cs]. [Online]. Available: http://arxiv.org/abs/2210.03629

[5] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. Providence RI USA: ACM, Apr. 2019, pp. 3–18. [Online]. Available: https://dl.acm.org/doi/10.1145/3297858.3304013

[6] P. Belcak, G. Heinrich, S. Diao, Y. Fu, X. Dong, S. Muralidharan, Y. C. Lin, and P. Molchanov, "Small Language Models are the Future of Agentic AI," Jun. 2025, arXiv:2506.02153 [cs]. [Online]. Available: http://arxiv.org/abs/2506.02153

[13] grafana.com/oss/loki
[14] www.firecrawl.dev