LLM-Based Multi-Agent Framework For Troubleshooting Distributed Systems

Mario De Jesus*, Perfect Sylvester†, William Clifford†, Aaron Perez†, and Palden Lama†
*Department of Electrical and Computer Engineering
The University of Texas at San Antonio, San Antonio, Texas, USA
mario.dejesus@my.utsa.edu
†Department of Computer Science
The University of Texas at San Antonio, San Antonio, Texas, USA
{perfect.sylvester, william.clifford, aaron.perez}@my.utsa.edu, palden.lama@utsa.edu

***Abstract*—Effectively configuring distributed systems, particularly those orchestrated by Kubernetes, remains challenging due to inherent complexity. This paper introduces KubeLLM, an LLM-based multi-agent framework designed to automate the troubleshooting of Kubernetes clusters. KubeLLM aims to diagnose configuration errors, analyze root causes, and apply necessary fixes, thereby reducing manual effort and improving system reliability. Our collaborative agents utilize Linux shell commands, remember interaction history, and access domain-specific knowledge via Retrieval Augmented Generation (RAG). We also introduce KubeLLMBench, a benchmark suite for evaluating LLM agents on Kubernetes troubleshooting tasks. Extensive evaluations conducted on a testbed emulating multi-node deployments demonstrate the feasibility and effectiveness of KubeLLM. We compare various LLMs (including Llama 3.3, OpenAI GPT-4o, Google Gemini 1.5 Flash, and o3-mini) across different agent configurations (single vs. multi-agent, memory vs. no memory, self-evaluation vs. no self-evaluation), analyzing performance in terms of accuracy, execution time, cost, and robustness against task-specific failures. Results show that multi-agent approaches generally improve average accuracy. While faster models risk complete failure on certain tasks, more robust models like o3-mini offer higher reliability. Notably, enabling self-evaluation reduces the likelihood of complete task failure, enhancing robustness at the cost of increased execution time. Specific configurations offer strong trade-offs between performance, speed, cost, and reliability, highlighting the critical need to balance these factors when deploying LLM agents in modern DevOps workflows.**

***Index Terms*—Kubernetes, Large Language Models (LLM), Multi-Agent Systems, Automated Troubleshooting.**

I. Introduction

Distributed systems, orchestrated by platforms like Kubernetes [1], are foundational to modern cloud-native applications. However, their configuration, deployment, and maintenance are complex and error-prone [2]. Misconfigurations in resources like Pods, Services, or Ingress controllers can lead to downtime, performance degradation, and significant debugging effort. Static analysis tools like Kube-Score [3] and KubeLinter [4] check Kubernetes manifests for pre-deployment issues against best practices. However, they cannot diagnose runtime errors, dynamic misconfigurations between components, or problems outside the manifests (e.g., application code, Docker files, environment variables, etc.).

Large Language Models (LLMs) have shown remarkable capabilities in understanding context, generating human-like text, and performing reasoning tasks. While LLMs show promise in general software repair [5]–[7], existing methods often target code-level bugs. However, applying these approaches to distributed systems orchestrated by Kubernetes is challenging. They may lack specific knowledge, hallucinate solutions, or struggle with multi-step reasoning and tool execution. Furthermore, selecting the optimal LLM model and agent configuration (e.g., single vs. multi-agent, memory use, self-correction mechanisms) is crucial for achieving reliable and efficient performance [8]. Recent studies [2], [9] have explored the use of LLMs to detect and suggest mitigations limited to security-related Kubernetes misconfigurations, but they still rely on human intervention to implement the recommended fixes. To our knowledge, this paper presents the first integrated framework and benchmark for autonomous LLM agents targeting both the automatic diagnosis and remediation of diverse operational configuration errors within Kubernetes.

Our key contributions are as follows:

- We designed and implemented KubeLLM, a flexible multi-agent framework for troubleshooting Kubernetes issues. The Knowledge Agent processes user queries, leveraging an LLM and Retrieval Augmented Generation [10] (RAG)-enhanced knowledge to formulate recommended actions. The Tools Agent then takes these recommendations, using its LLM to perform concrete actions like file modifications and ‘kubectl‘ commands directly on the Kubernetes cluster.
- We developed KubeLLMBench to evaluate LLM agent troubleshooting on diverse Kubernetes con-

2025 IEEE Cloud Summit | 979-8-3315-2362-6/25/$31.00 ©2025 IEEE | DOI: 10.1109/CLOUD-SUMMIT64795.2025.00024

979-8-3315-2362-6/25/$31.00 ©2025 IEEE
DOI 10.1109/Cloud-Summit64795.2025.00024

figuration errors. The suite uses faulty manifests, Dockerfiles, and application code to simulate runtime issues in networking, resource definitions, and dependencies—problems often beyond the scope of static analysis.
- We conducted extensive experiments evaluating different LLMs (proprietary and open-weight) and agent configurations, providing insights into the trade-offs between accuracy, execution time, cost, and robustness against task failures.

Our results demonstrate that KubeLLM can effectively troubleshoot various Kubernetes problems, with multi-agent configurations significantly enhancing performance for most models.

II. Related Work

While significant research has focused on automating software debugging and repair at the code level [5]–[7], extending these efforts to distributed systems like Kubernetes is challenging.

A. Static Analysis Tools

Existing tools perform rule-based static analysis of Kubernetes manifests to identify potential issues before deployment. Kube-Score [3] and KubeLinter [4] analyze YAML files against best practices for security and reliability. They can detect issues like missing resource limits, incorrect privileges, or probe misconfigurations. **Limitation:** These tools struggle with adapting to new misconfigurations since they rely on static rules. Furthermore, they do not address problems originating outside the Kubernetes configuration files (e.g., in application code or underlying infrastructure). KubeLLM aims to address these dynamic, runtime problems.

B. LLM Agents for Code Repair

Recent research explores using LLMs for general software debugging and repair. RepairAgent [5] uses LLMs for autonomous program repair with dynamic planning. Conversational Automated Program Repair [6] introduces iterative feedback. FixAgent [7] employs a hierarchical multi-agent approach for debugging Java code, demonstrating the power of collaborative agents. **Limitation:** While promising, these approaches often focus on code-level bugs within specific programming languages. Applying them directly to the complexities of distributed system configuration and orchestration in Kubernetes requires specialized tools, domain knowledge, and system interaction capabilities, which KubeLLM provides. KubeLLM specifically targets Kubernetes runtime issues using system interaction tools and relevant documentation via RAG.

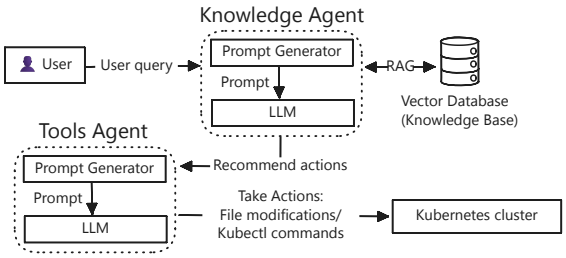


Fig. 1: KubeLLM Architecture

C. LLM Frameworks for Kubernetes Configuration

In recent works, Lanciao et al. [11] applied LLM-based methods for the detection of misconfiguration in Kubernetes Configuration Files (KCF). Minna et al. [9] further utilized LLM to suggest the mitigation of KCF misconfigurations. Malul et al. [2] developed a fine-tuned LLM that accurately identifies a broad range of KCF misconfiguration. Furthermore, they adapted a pre-trained LLM using prompt engineering and few-shot learning to locate each misconfiguration, explain it clearly, and suggest fixes. **Limitation:** While these approaches detect and suggest fixes, primarily for security misconfigurations, they lack agency, requiring human intervention for remediation. In contrast, KubeLLM employs autonomous LLM agents that not only diagnose but also fix a broader spectrum of general operational configuration errors, including issues in networking, resource definitions, and dependencies.

III. The KubeLLM Framework

KubeLLM is designed as a flexible framework to facilitate the use of LLM agents for Kubernetes troubleshooting. KubeLLM adopts a multi-agent paradigm [8], creating specialized agents for knowledge retrieval and tool execution within the Kubernetes context.

A. Architecture

Fig. 1 illustrates the KubeLLM multi-agent architecture. The Knowledge Agent leverages RAG and an LLM to interpret user queries and recommend actions, while the Tools Agent employs its LLM to execute these actions (file changes, kubectl commands) on the Kubernetes cluster.

Knowledge Agent: This agent is designed to detect and suggest fixes for configuration issues. Based on the user query (problem description) and relevant context (e.g., Kubernetes manifest files, Dockerfiles, application code), its Prompt Generator combines the relevant file contents with the user query to generate a prompt. Then, it feeds the prompt to a user-specified LLM, which formulates a plan and recommends a sequence of actions (shell commands, file modifications using

'sed', 'kubectl' commands) to fix the issue. To enhance the LLM's domain-specific expertise, it leverages Retrieval Augmented Generation (RAG) [10] to access a knowledge base containing Kubernetes documentation on common error patterns.

Tools Agent: This agent is designed to execute the corrective actions as recommended by the Knowledge Agent. First, its Prompt Generator wraps the message received from the Knowledge Agent with carefully crafted instructions to generate a prompt. Feeding the prompt to a user-specified LLM and leveraging a custom execution module, it directly executes various operations on the Kubernetes cluster. The operations include executing 'sed' commands to modify relevant files and issuing 'kubectl' commands to apply configuration changes, inspect logs, and monitor the status of cluster resources.

B. Key Features of KubeLLM

LLM Flexibility: Agents can be configured to use various LLMs, including local open-weight models (like Llama 3.3 [12]) or API-based proprietary models (like GPT-4o [13], Gemini 1.5 Flash [14], o3-mini [15]).

RAG Integration: Enhances the Knowledge Agent's understanding with domain-specific documentation, reducing reliance solely on the LLM's pre-trained knowledge and mitigating hallucination.

Tool Usage: Equips the Tools Agent with necessary capabilities (shell execution, 'kubectl') to interact with and modify the system state.

Memory: Both agents can be configured with memory to retain chat history across multiple turns within a single troubleshooting session. This allows for context preservation and iterative refinement of solutions.

Self-Evaluation: We implemented an optional mode where the Tools Agent executes one action at a time, evaluates the outcome (e.g., checking pod status), and makes real-time adjustments. This enables a more cautious, iterative approach compared to executing the entire plan at once.

C. KubeLLMBench Benchmark Suite

We developed KubeLLMBench, a suite of common Kubernetes troubleshooting scenarios. Each scenario involves specific misconfigured files (YAML manifests, Dockerfiles, application code). The framework automatically sets up the environment for each test case based on the configuration specified in its JSON file, runs the KubeLLM agent(s), and then tears down the environment. Key benchmark tasks include:

- Incorrect Selector (Pod label vs. Service selector mismatch)
- Wrong Port (Container port vs. application port mismatch)

```
DEBUG
The pod on my Kubernetes cluster cannot be accessed by making a curl request.
Any requests made to the application running in this pod result in a timeout.
A user has recently made a recent change to server.py.  Before the change everything
was working fine.  Give specific commands to fix this issue.
If modifying file contents is necessary, use the sed command to achieve this.
If providing a file path, make sure to keep the full file path.


The file /home/mario/KubeLLM_latest/KubeLLM-main/debug_assistant_latest/
troubleshooting/wrong_port/wrong_port.yaml
describes a deployment.  This is the file contents:
apiVersion: v1
kind: Pod
metadata:
  name: kube-wrong-port
spec:
  containers:
    - name: kube-wrong-port
      image: marioutsa/kube-wrong-port-app
      ports:
        - containerPort: 8765

The file /home/mario/KubeLLM_latest/KubeLLM-main/debug_assistant_latest/
troubleshooting/wrong_port/server.py
describes an application.  This is the file contents:
import http.server
import socketserver

Handler = http.server.SimpleHTTPRequestHandler

with socketserver.TCPServer(("", 8000), Handler) as httpd:
    httpd.serve_forever()

The file /home/mario/KubeLLM_latest/KubeLLM-main/debug_assistant_latest/
troubleshooting/wrong_port/Dockerfile
describes a Dockerfile.  This is the file contents:
FROM ubuntu:20.04
RUN apt-get update && \
    apt-get install -y \
        python3 \
        python3-pip \
        python3-venv \
        inotify-tools # Install inotify-tools for file change detection

WORKDIR /app
COPY server.py /app
EXPOSE 8765
CMD ["python3", "server.py"]
```

Fig. 2: Prompt fed to the Knowledge Agent

```
DEBUG   The issue seems to be caused by a port mismatch between the port
on which the
        server is running (8000)
        and the port that is exposed and used by the Pod specification
(8765).  We need to
        align the port numbers
        so that they match.


Here's what you should do:

1.  Update the server.py to use the port specified in the Pod definition
(8765):

  Run the following sed command to change the port in server.py from 8000
to 8765.

sed -i 's/8000/8765/' \
/home/mario/KubeLLM_latest/KubeLLM-main/debug_assistant_latest/troubleshooting/
    wrong_port/server.py

2.  Build the Docker image again with the new server configuration:

cd /home/mario/KubeLLM_latest/KubeLLM-main/debug_assistant_latest/troubleshooting/
    wrong_port
docker build -t marioutsa/kube-wrong-port-app .

3.  Update the Pod with the new Docker image:

kubectl apply -f \
/home/mario/KubeLLM_latest/KubeLLM-main/debug_assistant_latest/troubleshooting/
    wrong_port/wrong_port.yaml

4.  Verify the Pod is running correctly:

kubectl get pods | grep kube-wrong-port

5.  Check the logs for any errors:

kubectl logs kube-wrong-port

After these steps, the application should be accessible via the expected
port.
```

Fig. 3: Response received from the Knowledge Agent

- Liveness Probe Misconfiguration (Incorrect path/port)
- Wrong Interface (Server binding to 'localhost' instead of '0.0.0.0')
- Port Mismatch (Inconsistent ports across Pod, Service, Dockerfile)
- Misspelling (e.g., in image name or labels)
- Invalid Volume Mount
- Missing Environment Variable

As an illustrative example, Fig.2 presents the prompt provided to the Knowledge Agent's LLM for the Wrong Port scenario, while Fig.3 displays the corresponding response. Subsequently, the code snippet in Listing 1 illustrates how the Knowledge Agent's recommended actions are wrapped with additional instructions to construct a prompt for the Tool Agent's LLM. Furthermore, when self-evaluation mode is enabled, a specially designed wrapper instruction is applied to each recommended action step-by-step, as illustrated in Listing 2.

D. Implementation

We implemented KubeLLM using Phidata (Agno) [16], an open-source library for building AI agents. To support RAG, we implemented the knowledge base required by the Knowledge Agent using pgvector, an open-source PostgreSQL extension that allows storage and vector embeddings search in a PostgreSQL database. The knowledge source included Kubernetes documentation on common error patterns. To enable the Tools Agent to take actions, we implemented a custom execution module-—an enhanced version of Phidata's default shell tool. The implementation of the KubeLLM framework and the KubeLLMBench evaluation suite, consisting of approximately 2,300 lines of Python code along with benchmark configurations, is publicly available on GitHub [17].

IV. Experimental Setup

A. Testbed Environment

To evaluate KubeLLM using various configurations and LLMs, we conducted our experiments on a physical server hosting Docker containers, emulating multiple compute nodes. Kubernetes v1.31.5 was deployed locally using Minikube [18] for container orchestration. The physical server was equipped with 2 x Intel Xeon Gold 6338 (32 cores each) CPU, 768 GB RAM and 2 x Nvidia A40 GPUs (46GB VRAM each). The host operating system was Ubuntu Linux Server 20.04.

B. LLMs and Agent Configurations

We evaluated KubeLLM using four LLMs—Gemini 1.5 Flash [14], GPT-4o [13], Llama 3.3 (70B, via Ollama) [12], and o3-mini [15]. These models were selected to cover a range of categories, including open-weight vs. proprietary, reasoning-optimized vs. lightweight, and local vs. API-based execution. We evaluated six agentic interaction modes: Single-Agent (with and without memory) and Multi-Agent (with and without memory, each with optional self-evaluation mode). Each combination of LLM and agentic interation modes was tested on all KubeLLMBench tasks with multiple trials (e.g., 10 runs / task) to ensure consistency of the results.

```
prompt = f'Perform the action suggested here: \n{self.agentAPIResponse}\n'
prompt += f"\nThe relevant configuration file is located in this path: {self.config['test-directory'] + self.config['yaml-file-name']}\n"
prompt += "You can update these files if necessary. If any files are updated, make sure to delete and reapply the configuration file.\n"
```

Listing 1: Wrapper instructions applied to the Knowledge Agent's response, {self.agentAPIResponse}.

```
for i, step in enumerate(self.steps, start=1):
    prompt = f'Perform the action suggested here: \n{step}\n'
    prompt += f'If you struggle within one of the steps try to figure out the solution until you see the pod running fine with kubectl describe.'
    prompt += f"\nThe relevant configuration file is located in this path: {self.config['test-directory'] + self.config['yaml-file-name']}\n"
    prompt += "You can update these files if necessary. If any files are updated, make sure to delete and reapply the configuration file.\n"
    prompt += "If you need to update a pod then use kubectl replace --force [POD_NAME]"
```

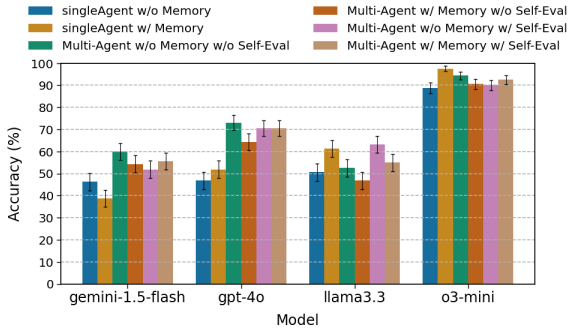
Listing 2: Wrapper instructions applied when self-evaluation option is enabled. Here, {step} refers to one action from a sequence of recommended actions.

C. Evaluation Metrics

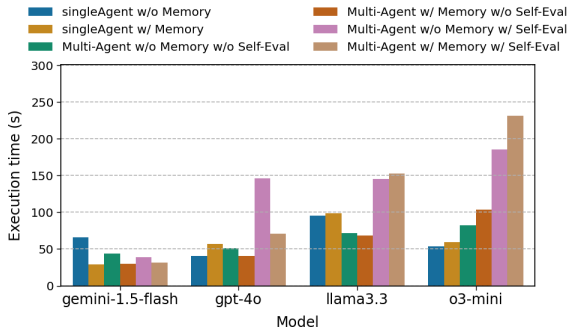
Performance was evaluated using two metrics: (1) Accuracy—the percentage of runs where KubeLLM successfully resolved the issue, verified by examining the final configuration files and ensuring the Kubernetes pod reached the Running state; and (2) Average Execution Time—the mean duration (in seconds) from the initial prompt to the agent's final action, regardless of success. Failed runs were recorded along with agent-provided explanations (if any) for qualitative analysis.

V. Results

Figures 4a and 4b present the average accuracy (i.e., success rate) and execution time of KubeLLM across the various tasks outlined in Section III-C, evaluated using four different LLMs and six agent configurations. In all multi-agent configurations, the LLM assigned to the Knowledge Agent was consistently set to o3-mini,



(a) Average accuracy (success rate)



(b) Average execution time

Fig. 4: KubeLLM performance with various models and agent configurations. o3-mini is used as Knowledge Agent in all multi-agent configurations.

while various models were used for the Tools Agent. This design choice is motivated by the hypothesis that employing a stronger reasoning model to generate plans for a cheaper (and less capable) model to execute may strike a favorable balance between performance and cost.

Table I presents a comparison of the best-performing agent configurations based on average accuracy, execution time, and estimated cost. The cost is computed using the total number of input and output tokens consumed by each model, multiplied by their respective API pricing. For reference, the API pricing per 1 million tokens is as follows: o3-mini — $1.10 (input) and $1.40 (output); gpt-4o — $2.50 (input) and $10.00 (output); and gemini-1.5-flash — $0.075 (input) and $0.30 (output). The open-weight model llama3.3, hosted locally, incurs no cost. Table II provides a detailed breakdown of performance at task-level granularity for the various configurations.

A. Key Observations

Our experiments reveal significant performance differences across LLMs and agent configurations, highlighting critical trade-offs between accuracy, execution time, cost, and, robustness against specific task failures.

TABLE I: Comparing the average accuracy, speed, and cost of best-performing agent configurations.

Agent Config.	Model(s)	Accuracy (%)	Execution Time (s)	Cost ($)
single w/ mem	o3-mini	**97.5%**	59.19s	$0.02
multi-agent w/o mem	o3-mini & gpt-4o	73.12%	51.18s	$0.029
multi-agent w/o mem w self-eval	o3-mini & llama3.3	63.12%	145s	**$0.006**
multi-agent w/o mem	o3-mini & gemini-1.5-flash	60%	**43.98s**	$0.008

Accuracy vs. Speed vs. Cost Trade-offs Table I highlights the following key observations:

- **Single Agent o3-mini (w/ Memory)** shows high average accuracy (97.5%) and reasonable speed (59.19s) at moderate cost ($0.02) per task.
- **Multi-Agent o3-mini & Gemini 1.5 Flash (w/o Memory)** is fastest (43.98s) and cheap ($0.008) but has low average accuracy (60%).
- **Multi-Agent o3-mini & Llama 3.3 (w/o Memory, w/ Self-Eval)** is cheapest ($0.006) but slow (145s) with 63.12% accuracy.
- **Multi-Agent o3-mini & GPT-4o (w/o Memory)** offers competitive speed (51.18s) at the highest cost ($0.029) with average accuracy of 73.12%.

Task-Specific Failures and Robustness A critical observation from Table II is the occurrence of **complete failure (0% accuracy)** on specific tasks for certain configurations. Models like **Gemini 1.5 Flash** frequently scored 0% on tasks like 'Env Var' and 'Liveness Pr.' when self-evaluation was not employed, indicating potential blind spots. Models like **o3-mini** demonstrated greater inherent robustness, rarely hitting 0% regardless of configuration. GPT-4o showed intermediate robustness. This highlights that average accuracy can mask critical weaknesses; the risk of complete failure on certain task types is worth considering.

Impacts of Agent Configuration

- **Multi-Agent setups** generally improve accuracy.
- **Memory** offers limited benefit.
- **Self-Evaluation** significantly increases execution time across all models. However, Table II shows that enabling self-evaluation **reduces the likelihood of complete task failure (0% accuracy)**. Comparing configurations with and without self-evaluation for the same base models, the self-evaluation mode often achieves better accuracy on tasks where the non-self-evaluating counterpart failed completely. This enhanced robustness comes at the cost of increased execution time.

TABLE II: Performance of Various Models and Approaches at Task-Level Granularity (Accuracy % / Avg. Time s)

Approach	Model	Env Var	Incorrect Sel.	Liveness Pr.	Misspell	Port Mism.	Volume Mnt.	Wrong Iface.	Wrong Port
Multi-Agent w/o Memory w/o Self-Eval	gemini-1.5-flash	0.0%/19.4s	95.0%/29.5s	0.0%/95.5s	55.0%/22.2s	95.0%/32.4s	55.0%/22.8s	100.0%/83.7s	80.0%/46.4s
	gpt-4o	85.0%/93.6s	100.0%/35.0s	0.0%/20.4s	70.0%/24.4s	90.0%/36.8s	55.0%/22.9s	100.0%/83.6s	85.0%/92.9s
	llama3.3	0.0%/66.4s	100.0%/64.7s	0.0%/70.6s	35.0%/58.8s	100.0%/88.3s	25.0%/71.0s	70.0%/68.1s	90.0%/84.1s
	o3-mini	95.0%/103.4s	90.0%/100.4s	100.0%/88.4s	75.0%/48.5s	100.0%/68.7s	95.0%/79.6s	100.0%/78.1s	100.0%/88.1s
Multi-Agent w/o Memory w/ Self-Eval	gemini-1.5-flash	10.0%/20.9s	90.0%/35.1s	25.0%/28.5s	20.0%/30.6s	100.0%/26.8s	50.0%/45.5s	40.0%/53.2s	80.0%/68.7s
	gpt-4o	70.0%/116.3s	95.0%/109.8s	75.0%/107.9s	55.0%/82.6s	100.0%/107.3s	40.0%/60.3s	80.0%/384.3s	50.0%/202.3s
	llama3.3	0.0%/104.8s	95.0%/183.3s	25.0%/119.5s	70.0%/177.4s	95.0%/99.3s	55.0%/120.4s	75.0%/172.8s	90.0%/183.1s
	o3-mini	90.0%/104.7s	85.0%/87.5s	100.0%/313.4s	85.0%/175.1s	95.0%/121.3s	100.0%/123.3s	80.0%/245.0s	85.0%/312.9s
Multi-Agent w/ Memory w/o Self-Eval	gemini-1.5-flash	0.0%/17.3s	60.0%/38.4s	5.0%/18.3s	40.0%/19.0s	100.0%/26.2s	45.0%/17.6s	95.0%/54.6s	90.0%/49.5s
	gpt-4o	45.0%/34.7s	100.0%/17.2s	0.0%/31.4s	60.0%/46.6s	100.0%/16.1s	25.0%/28.0s	100.0%/97.9s	85.0%/50.1s
	llama3.3	0.0%/61.2s	0.0%/60.7s	5.0%/69.6s	45.0%/63.3s	100.0%/73.5s	45.0%/66.1s	80.0%/68.9s	100.0%/85.1s
	o3-mini	95.0%/97.9s	75.0%/140.9s	90.0%/93.5s	90.0%/70.8s	100.0%/86.3s	75.0%/102.5s	100.0%/89.6s	100.0%/144.8s
Multi-Agent w/ Memory w/ Self-Eval	gemini-1.5-flash	0.0%/17.1s	75.0%/21.6s	40.0%/18.5s	30.0%/29.2s	100.0%/14.6s	55.0%/20.1s	55.0%/54.3s	90.0%/80.3s
	gpt-4o	60.0%/38.4s	80.0%/27.4s	65.0%/125.0s	70.0%/66.0s	95.0%/59.7s	60.0%/42.4s	70.0%/104.3s	65.0%/105.9s
	llama3.3	0.0%/144.9s	65.0%/163.3s	20.0%/113.9s	60.0%/224.9s	100.0%/130.6s	40.0%/89.3s	55.0%/183.7s	100.0%/169.5s
	o3-mini	95.0%/321.3s	85.0%/73.8s	100.0%/362.1s	80.0%/315.8s	100.0%/105.7s	100.0%/190.0s	80.0%/230.1s	100.0%/255.1s
Single-Agent w/o Memory	gemini-1.5-flash	0.0%/172.2s	100.0%/4.3s	65.0%/36.4s	0.0%/6.1s	95.0%/4.3s	100.0%/253.6s	0.0%/7.3s	10.0%/42.9s
	gpt-4o	0.0%/26.6s	25.0%/16.1s	80.0%/49.1s	25.0%/19.4s	100.0%/90.1s	100.0%/44.7s	45.0%/39.2s	0.0%/37.3s
	llama3.3	5.0%/105.2s	95.0%/61.0s	90.0%/117.2s	15.0%/102.9s	95.0%/122.9s	100.0%/74.4s	0.0%/73.1s	5.0%/107.8s
	o3-mini	100.0%/30.0s	100.0%/55.6s	100.0%/57.0s	100.0%/54.2s	100.0%/52.7s	100.0%/59.8s	100.0%/63.7s	10.0%/59.0s
Single-Agent w/ Memory	gemini-1.5-flash	0.0%/7.3s	45.0%/76.0s	60.0%/17.4s	0.0%/4.2s	100.0%/4.3s	100.0%/104.5s	0.0%/5.3s	5.0%/16.2s
	gpt-4o	10.0%/152.8s	50.0%/33.2s	95.0%/39.7s	0.0%/27.0s	100.0%/48.0s	95.0%/51.3s	65.0%/53.0s	0.0%/52.3s
	llama3.3	0.0%/88.6s	90.0%/49.2s	100.0%/104.4s	0.0%/150.6s	100.0%/101.2s	100.0%/165.7s	0.0%/63.3s	100.0%/66.1s
	o3-mini	100.0%/42.4s	90.0%/77.3s	100.0%/52.5s	95.0%/58.3s	95.0%/59.6s	100.0%/62.2s	100.0%/70.7s	100.0%/50.5s

In summary, faster/cheaper models carry a higher inherent risk of complete failure on certain tasks. Self-evaluation acts as a mitigation mechanism, reducing the probability of these 0% accuracy scenarios but imposing a significant latency penalty. Reasoning models like o3-mini are inherently more robust, potentially making the time cost of self-evaluation less justifiable for them.

VI. Conclusion

This paper introduced KubeLLM, a multi-agent LLM framework for automated Kubernetes troubleshooting, and KubeLLMBench for its evaluation. We demonstrated the feasibility of using LLM agents with tools and RAG to autonomously diagnose and resolve common Kubernetes configuration errors. Our comparative analysis of different LLMs (GPT-4o, Gemini 1.5 Flash, Llama 3.3, o3-mini) and agent configurations revealed significant trade-offs between average accuracy, speed, cost, and crucially, robustness against task-specific failures.

Key findings indicate that while faster models like Gemini excel in speed, they risk complete failure (0% accuracy) on certain tasks. More robust models like o3-mini demonstrate higher reliability across diverse problems but are slower. Multi-agent architectures generally improve average accuracy. Notably, self-evaluation, while significantly increasing execution time, was observed to reduce the likelihood of complete task failure, offering a trade-off between speed and robustness. Selecting the optimal LLM and agent configuration requires careful consideration of operational priorities—balancing the acceptable risk of failure against the needs for accuracy, speed, and cost.

Acknowledgment

This material is based upon work supported by the U.S Department of Energy, Office of Science, Office of Advanced Scientific Computing Research (ASCR), Reaching a New Energy Sciences Workforce (RENEW) under Award Number DE-SC0024322, and the UTSA School of Data Science UG Research Fellowship.

References

[1] "Kubernetes," https://kubernetes.io.
[2] E. Malul, Y. Meidan, D. Mimran, Y. Elovici, and A. Shabtai, "Genkubesec: Llm-based kubernetes misconfiguration detection, localization, reasoning, and remediation," *arXiv preprint arXiv:2405.19954*, 2024.
[3] "Kube-score," https://github.com/zegl/kube-score.
[4] "Kubelinter," https://github.com/stackrox/kube-linter.
[5] I. Bouzenia, P. Devanbu, and M. Pradel, " RepairAgent: An Autonomous, LLM-Based Agent for Program Repair ," in *IEEE/ACM Int'l Conf. on Software Engineering (ICSE)*, 2025.
[6] C. S. Xia and L. Zhang, "Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using chatgpt," in *ACM SIGSOFT Int'l Symposium on Software Testing and Analysis (ISSTA)*, 2024.
[7] C. Lee, C. S. Xia, L. Yang, J.-t. Huang, Z. Zhu, L. Zhang, and M. R. Lyu, "A unified debugging approach via llm-based multi-agent synergy," *arXiv preprint arXiv:2404.17153*, 2024.
[8] K.-T. Tran, D. Dao, M.-D. Nguyen, Q.-V. Pham, B. O'Sullivan, and H. D. Nguyen, "Multi-agent collaboration mechanisms: A survey of llms," *arXiv preprint arXiv:2501.06322*, 2025.
[9] F. Minna, F. Massacci, and K. Tuma, "Analyzing and mitigating (with llms) the security misconfigurations of helm charts from artifact hub," *arXiv preprint arXiv:2403.09537*, 2024.
[10] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," *Advances in Neural Information Processing Systems*, vol. 33, pp. 9459–9474, 2020.
[11] G. Lanciano, M. Stein, V. Hilt, T. Cucinotta *et al.*, "Analyzing declarative deployment code with large language models," *CLOSER*, vol. 2023, pp. 289–296, 2023.
[12] "Ollama llama 3," https://ollama.com/library/llama3.
[13] OpenAI, "Gpt-4o," https://openai.com/index/hello-gpt-4o/.
[14] "Google gemini," https://deepmind.google/technologies/gemini/flash/.
[15] "Openai o3-mini," https://openai.com/index/openai-o3-mini/.
[16] "Phidata," https://www.phidata.ai.
[17] M. D. Jesus, P. Sylvester, W. Clifford, A. Perez, and P. Lama, "KubeLLM," https://github.com/cloudsyslab/KubeLLM.
[18] "Minikube," https://minikube.sigs.k8s.io/.