POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Beyond AI in the Wild: Optimizing Root Cause Analysis with Multi-Agent Systems

Tesi di Laurea Magistrale in
Computer Science and Engineering - Ingegneria Informatica

Marcello Martini, 243880

Abstract: The transition from monolithic architectures to distributed microservices has obscured system visibility, making manual incident response a significant bottleneck for Site Reliability Engineering (SRE) teams. Although Large Language Models (LLMs) are increasingly explored for Root Cause Analysis (RCA), deploying them introduces "AI in the Wild" challenges, including high token costs, context saturation, and hallucinations.
To address these limitations, this thesis introduces a novel Multi-Agent System (MAS) for RCA in Kubernetes environments. The architecture employs a divide and conquer strategy, decomposing complex diagnostics into manageable sub-tasks executed by specialized agents. Key contributions include a hybrid deterministic-generative Triage agent, a Datagraph to ground reasoning in the cluster topology, and a custom Model Context Protocol (MCP) server that distills telemetry data to optimize token efficiency and mitigate hallucinations.
Experimental validation in 17 fault scenarios demonstrated that this approach, using the cost-effective GPT-5-mini model, outperformed state-of-the-art baselines (AIOpsLab using GPT-4). The system achieved a Localization Accuracy of 82% versus 61.54% for AIOpsLab, corresponding to a +20.46 percentage-point gain (i.e., a +33.25% relative improvement). In addition, a within-model comparative analysis across GPT-5-mini configurations shows that Planner-guided tool selection yields statistically significant efficiency gains, reducing token usage by up to 42% and execution time by up to 29%. These findings suggest that architectural guardrails and task decomposition can outweigh raw model parameter counts for RCA, achieving strong performance with a lower-cost model; although parameter counts are not publicly disclosed for proprietary models, GPT-4 is widely regarded as having more parameters than GPT-5-mini.

Advisor:
Prof. Marco Domenico Santambrogio

Co-advisors:
Dr. Matteo Ferroni
Dr. Beatrice Branchini

Academic year:
2024-2025

Key-words: AIOps, Large Language Models, Multi-Agent Systems, Kubernetes, Root Cause Analysis, Site Reliability Engineering

Abstract in lingua italiana

La transizione dalle architetture monolitiche ai microservizi distribuiti ha ridotto la visibilità complessiva del sistema, trasformando la risposta manuale agli incidenti in un collo di bottiglia critico per le operazioni di Site Reliability Engineering (SRE). Sebbene i Large Language Models (LLM) stiano emergendo come strumenti promettenti per la Root Cause Analysis (RCA), la loro adozione in ambienti reali introduce le sfide tipiche della cosiddetta "AI in the Wild", quali costi elevati, saturazione della finestra di contesto e allucinazioni.
Per superare tali limitazioni, il presente lavoro propone un innovativo Sistema Multi-Agente (MAS) per la RCA in ambienti Kubernetes. L'architettura adotta una strategia di Divide et Impera, che scompone la complessità diagnostica in sotto-task gestiti da agenti specializzati. I contributi principali includono un agente di Triage ibrido (deterministico-generativo), l'impiego di un Datagraph per ancorare il ragionamento alla topologia del cluster e un server Model Context Protocol (MCP) appositamente sviluppato per filtrare e sintetizzare i dati di telemetria, ottimizzando così l'efficienza dei token e mitigando il rischio di allucinazioni.
La campagna sperimentale, condotta su 17 scenari di guasto, ha dimostrato che l'approccio proposto, utilizzando il modello cost-effective GPT-5-mini, supera le prestazioni delle soluzioni allo stato dell'arte (come AIOpsLab basato su GPT-4). Il sistema ha raggiunto un'accuratezza di localizzazione dell'82% contro il 61.54% di AIOpsLab, corrispondente a un guadagno assoluto di +20.46 punti percentuali (ossia un +33.25% di miglioramento relativo). Inoltre, un'analisi comparativa interna sulle configurazioni di GPT-5-mini evidenzia che la selezione degli strumenti guidata dal Planner produce guadagni di efficienza statisticamente significativi, riducendo l'uso dei token fino al 42% e il tempo di esecuzione fino al 29%. Tali risultati suggeriscono che l'adozione di guardrail architetturali e la scomposizione dei task possano prevalere sul mero numero di parametri del modello per la RCA, ottenendo prestazioni elevate con modelli a basso costo; sebbene i conteggi dei parametri non siano pubblici per i modelli proprietari, GPT-4 è ampiamente considerato avere un numero di parametri superiore rispetto a GPT-5-mini.

Parole chiave: AIOps, Large Language Models, Sistemi Multi-Agente, Kubernetes, Root Cause Analysis, Site Reliability Engineering

1. Introduction

The transition from monolithic architectures to **distributed microservices** has changed the operational landscape for Site Reliability Engineering (SRE) teams. Although distributed systems offer scalability, they obscure visibility; a single user request often involves dozens of services, creating an intricate network of interdependencies that is difficult to map manually. This structural complexity has made traceability and observability a prerequisite for effective troubleshooting in large-scale distributed systems [1]. Consequently, incident response has evolved from checking a single server's logs to the challenging task of correlating heterogeneous data (metrics, application logs, and distributed traces) across the entire infrastructure. This manual correlation is a primary bottleneck, increasing the **Mean Time To Resolution** (MTTR) and requiring significant human effort. Fundamentally, SRE methodology prioritizes the minimization of downtime and the mitigation of operational risk through data-driven observability [2].

To address this, the industry (led by major players like Microsoft and IBM) is investigating Artificial Intelligence not just for anomaly detection, but also for Root Cause Analysis (RCA) and mitigation. Large Language Models (LLMs) are particularly well-suited for this domain because observability data are largely textual (logs, configuration files, code). By combining LLMs with an agentic architecture, the model ceases to be a passive text processor and becomes a reasoning engine capable of querying tools and exploring the system state autonomously.

However, moving from theoretical agentic prototypes to production-grade RCA tools introduces distinct engineering challenges, often referred to as the **"AI in the Wild" problem**. The deployment of probabilistic agents in deterministic infrastructure environments reveals critical limitations in current approaches.

The primary problem is **inefficient resource usage**. Many existing solutions rely on "brute-forcing" the diagnosis using flagship, pay-per-use models (e.g., GPT-4). These approaches often feed the agent excessive data without optimization, resulting in high latency and prohibitive token costs [3] that are unsustainable for continuous, real-time monitoring. Closely related to this inefficiency is the issue of **context saturation** [4]. Providing an agent with raw, unfiltered telemetry often fails, leading to the **"Garbage In, Garbage Out" phenomenon**. When an LLM's context window is flooded with high-volume noise (such as raw distributed traces), the signal-to-noise ratio drops, causing the agent to ignore relevant anomalies and increasing the likelihood of hallucinations or misinterpretation of the system state.

Finally, an RCA agent cannot rely only on probabilistic reasoning. Certain diagnostic steps, such as verifying if a metric exceeds a specific threshold or checking a pod's status, require **deterministic checks**, not semantic interpretation. This aligns with modular approaches that combine language models with external components and explicit reasoning modules, rather than relying on a single monolithic prompt [5]. However, "wild" agents often lack these architectural guardrails. While foundational work confirms that coupling generation with external tools enhances reliability [6], most current implementations still rely on fragile, **custom-built interfaces**. To enable production-grade reliability, it is therefore critical to move beyond these ad-hoc integrations and adopt standardized protocols like the **Model Context Protocol (MCP)**.

1.1. Thesis Objective and Contributions

Given the limitations of unconstrained LLM reasoning, specifically the lack of guardrails and the tendency towards nondeterminism, the primary objective of this research is to develop architectural strategies that mitigate hallucinations while optimizing accuracy, execution time, and token efficiency.

To this end, this thesis introduces a novel **Multi-Agent System (MAS)** for Root Cause Analysis in Kubernetes environments, specifically designed to address the "AI in the Wild" challenges through the following core contributions:

- **divide and conquer architecture:** We propose a multi-agent framework that decomposes the complex RCA task into smaller, manageable sub-tasks. These are executed in parallel by specialized agents and subsequently aggregated, enforcing a structured, deterministic workflow over a purely probabilistic one.
- **hybrid deterministic-LLM grounding:** We introduce a hybrid approach to ground the agent's reasoning. A preliminary triage phase employs deterministic heuristics to identify anomaly metrics and pod failures. Furthermore, we implement a *Datagraph*, a topological representation of the microservices network, to map service interactions and infrastructure dependencies. This graph constrains the agent's planning space, ensuring that diagnostic actions are strictly relevant to the dependency chain of the affected service.
- **optimized tooling via MCP:** To tackle the context saturation problem, we present a custom **Model Context Protocol (MCP)** server. This server acts as an abstraction layer between the agents and the cluster monitoring tools. Instead of returning raw data, the MCP tools distill telemetry data at the source, returning only concise, high-signal information to the agent, thereby optimizing token usage and maintaining context clarity.

To validate the proposed system, we conducted an experimental campaign across two distinct testbed applications, covering a total of **17 fault scenarios**. A key aspect of our evaluation is the use of a cost-effective, compact model (GPT-5-mini) rather than a flagship large language model. This choice serves to demonstrate that with effective task decomposition and tool grounding, architectural design is a more significant determinant of RCA success than raw model parameter count. Finally, we performed a statistical hypothesis test to quantitatively prove how specific agent configurations significantly impact the system's efficiency. The proposed work is open source and available on GitHub [7].
The remainder of this thesis is structured as follows. Section 2 introduces the foundational concepts and reviews the current state of the art in AIOps and LLM-based diagnosis. Section 4 details the proposed methodology, describing the Multi-Agent System architecture, the Datagraph implementation, and the MCP Server design. Subsequently, Section 5 presents the experimental framework, including the testbed applications, the evaluation pipeline, and the agent configurations tested. Section 6 provides a comprehensive analysis of the experimental results, including the comparative benchmarking against state-of-the-art baselines. Finally, Section 7 summarizes the key findings, discusses the limitations of the current approach, and outlines potential directions for future research.

2. Background

In this section, we outline the theoretical foundations and technological landscape underlying this work. We begin by introducing the operational context, defining Site Reliability Engineering (Section 2.1) and the paradigm shift towards AIOps (Section 2.2). Subsequently, we describe the core enabling technologies, focusing on Large Language Models (Section 2.3) and the emergence of Agentic AI (Section 2.4). The discussion then moves to the specific architectural components adopted in the proposed system, including agentic reasoning patterns (Section 2.5) and the Model Context Protocol (Section 2.6).

2.1. Site Reliability Engineering

Site Reliability Engineering(SRE) [2, 8, 9] is a discipline within software engineering that focuses on IT Operations. Specifically, SRE teams deploy and use software to manage systems, solve problems, and automate operational tasks. The primary goal is to improve the reliability of the system by adopting a set of best practices in both the deployment and the monitoring of the infrastructure. This ensures adherence to Service Level Agreements (SLAs), formal contracts that define the availability commitments of a system and the expected time for remediation in the event of downtime.
Another critical responsibility of SRE teams is **handling production incidents** for critical systems. This involves identifying the Root Cause of the problem and executing a mitigation plan to fix the issue and restore the system. The resolution of production incidents must be as fast as possible, as downtime significantly impacts both business operations and the defined SLAs.
With the shift from monolithic architectures to microservices, the SRE role has become increasingly complex. Teams must now debug distributed infrastructures composed of different resources that interact to provide services. Consequently, a holistic view is crucial to properly identify the Root Cause of an incident and reduce the Mean Time to Resolution.

2.2. AIOps

Artificial Intelligence for IT Operations (AIOps) [10–13] is the application of Artificial Intelligence to streamline and optimize IT operations. As modern microservice architectures generate massive volumes of observability data, specifically Metrics, Events, Logs, and Traces, traditional manual monitoring has become unfeasible. AIOps addresses this by enabling, for example, the ability to query infrastructure status using natural language or employing predictive models to forecast resource saturation and prevent incidents.
One of the primary tasks where AIOps is applied is incident management. Historically, this relied on statistical anomaly detection and log clustering. However, nowadays, Large Language Models (LLMs) have enabled the analysis of vast amounts of unstructured text to recognize complex patterns. This capability allows for a **holistic view** of the entire cloud infrastructure, significantly accelerating incident response.
Increasingly, companies are investing in AI applications for incident management with two primary objectives: **RCA**, which identifies the affected resource and underlying cause to accelerate mitigation; and **automated mitigation**, which utilizes AI to actively resolve incidents. While the first approach serves as a decision-support tool for SRE teams, the second aims to operate on the cluster with minimal human intervention, moving towards the concept of **self-healing systems**.

2.3. Large Language Models

The primary reasoning engines used for emerging AIOps applications are Large Language Models (LLMs) [14, 15]. These are generative language models trained on large amount of text using a self-supervised approach. Deep learning models of this class are composed of a massive number of parameters, often in the order of tens of billions, and are specifically designed for Natural Language Processing (NLP) tasks, particularly text generation.
The underlying architecture is the **Transformer**, introduced by Vaswani et al. [16], which serves as the foundational block for modern language models. This architecture introduced the self-attention mechanism alongside an encoder-decoder structure to facilitate effective text generation. Leveraging this discovery, several companies began developing proprietary models based on Transformers. Notable examples include OpenAI, which released models such as GPT-2 [17], an open-source model released in 2019 and one of the first to utilize this architecture successfully, and its subsequent iterations.
Selecting the appropriate LLM for a specific task involves evaluating several factors. First is cost: LLMs are generally divided into open-weight models, which can be downloaded and run locally given sufficient hardware, and closed-weight models (such as those from OpenAI or Google), which are accessed via API on a pay-per-use basis. Typically, larger models are more expensive, with costs calculated based on tokens, the atomic units of text used by the model for input and output processing.
Another crucial aspect is the **context window**, defined as the maximum amount of text (expressed in tokens) that the model can process in a single interaction. The context window acts as a form of working memory; since LLMs are stateless, all relevant information regarding the context or history of previous iterations must be provided within the input tokens. Finally, model selection depends on reasoning capabilities, often measured by standard benchmarks such as MMLU-Pro [18], as well as inference speed and multimodal capabilities which is the ability to process non-textual data types, such as images or audio, alongside text.
Despite their capabilities, LLMs have a significant downside: they suffer from **hallucinations** [19]. A hallucination is a phenomenon where the language model generates grammatically correct answers that are factually inconsistent, unsupported by training data, or completely fabricated.
To mitigate this issue, several techniques are used, ranging from **context engineering** (i.e., providing the model with all necessary context to answer correctly) and Retrieval-Augmented Generation (RAG) [20], to the implementation of guardrails and additional validation layers. These measures are essential to reduce the risks associated with deploying *"AI in the wild"* and to ensure reliability in production environments.

2.4. Agentic AI

In recent years, advancements in Large Language Models (LLMs) have gave rise to a new field known as Agentic AI [21]. Agentic AI refers to artificial intelligence systems capable of accomplishing specific goals with minimal human intervention. In these systems, the LLM serves as the central reasoning engine, equipped with a set of tools that allow the model to gather real-world data and execute actions similar to those of a human operator. A critical challenge in these applications is managing the context window, which can grow drastically during complex tasks. Therefore, a trade-off must be found between retaining information from previous iterations (memory) and conserving the token budget for the current context.
These agents can operate in two primary modes: **fully autonomous**, where no human intervention is required, or **human in-the-loop**, where a human provides explicit confirmation for sensitive actions or feedback to the system.
While a single agent combines an LLM with tools to manage a task autonomously, systems can also be composed of multiple agents, forming a **MAS**. In this architecture, each agent specializes in a specific sub-task, while an orchestrator supervises the agents' work and manages the overall workflow.

2.5. Agentic Patterns

With the advent of autonomous AI agents, several implementation patterns have been proposed to enhance reliability and simplify the orchestration of actions. An agentic pattern serves as an **architectural and behavioral template** guiding the development of AI systems, enabling agents to collaborate effectively with external tools, environments, and peer agents.
In the proposed architecture, two primary patterns are utilized:

- **Reason and Act (ReAct):** Proposed by Yao et al. (2023) [22], this pattern involves using an LLM to generate both **reasoning traces** and **task-specific actions** in an **interleaved manner**. At each iteration, the model invokes external tools to gather information or execute actions, waits for the output, and subsequently reasons upon the result to determine the next operational step. Furthermore, the

agent generates internal reasoning traces, textual annotations describing relevant insights or outlining necessary steps for subsequent iterations. While the agent's memory consists of the complete interaction history (providing full context), this accumulation can potentially saturate the context window, leading to hallucinations or excessive token consumption.

- **Plan-and-Execute:** Proposed by Wang et al. (2023) [23], this pattern aims to decompose complex objectives into a **structured sequence of sub-tasks**. Initially, the agent analyzes the problem to formulate a step-by-step execution strategy. Subsequently, each sub-task is executed, with the outputs often serving as inputs for the following steps. This approach is particularly advantageous in contexts where the objective requires numerous steps that would overwhelm a standard ReAct loop. By narrowing the scope of execution and requiring fewer iterations per sub-task, this pattern **mitigates hallucination** risks and keeps the context window size within manageable limits.

2.6. Model Context Protocol (MCP)

The **Model Context Protocol (MCP)** [24] is an open standard designed to facilitate seamless connectivity between AI applications and external systems. Introduced in late 2024 by Anthropic, this protocol addresses the interoperability challenges in the AI ecosystem by defining a **universal interface** for exposing tools, external knowledge repositories, and system prompts to AI agents, while supporting security mechanisms such as authentication.
The MCP ecosystem includes a suite of Software Development Kits (SDKs) that facilitate both the creation of MCP servers and their integration into LLM clients. Functionally, an MCP server exposes three distinct categories of capabilities:

- **Tools:** Enable the exposure of executable functions that the model can invoke to perform actions or computations within the external system.
- **Resources:** Provide access to structured data or content that can be read by clients and utilized as context to enrich LLM reasoning.
- **Prompts:** Define reusable templates that standardize common workflows and interaction patterns, allowing users to use pre-configured instructions.

Architecturally, a client can maintain simultaneous connections to multiple MCP servers. Communication is handled via standardized transport layers: Server-Sent Events (SSE) over HTTP (optimized for remote server connections) or stdio (optimized for local processes).

3. State of the Art

This section focuses our research on two fundamental areas of AIOps. We begin by examining the evaluation frameworks that enable reproducible benchmarking (Section 3.1), specifically focusing on fault injection and scoring protocols. From there, we analyze the current landscape of LLM-based systems (Section 3.2), highlighting the specific limitations and assumptions that our approach aims to overcome.

3.1. Evaluation Frameworks in AIOps

The rapid expansion of AIOps has driven the development of evaluation frameworks designed to assess AI agents on operational tasks. These tasks typically include Root Cause Analysis (RCA), mitigation planning, incident escalation management, and report generation. To support reproducible comparisons, several organizations have invested in standardized benchmarks, including IBM's *ITBench* and Microsoft's *AIOpsLab*.
Both frameworks are designed to evaluate agents operating within Kubernetes environments and generally share a modular architecture comprising four key components:

1. **Testbed Applications:** Configuration files (e.g., Helm charts) to deploy microservices alongside a comprehensive monitoring stack and workload generators.
2. **Fault Injector:** A mechanism to introduce controlled anomalies (faults) into the running testbed.
3. **Agent Interface:** A layer facilitating the AI agent's interaction with the cluster and observability tools (e.g., querying metrics or logs).
4. **Evaluator:** A module to quantify performance using metrics such as detection accuracy, execution time, and token consumption.

Two representative examples are *ITBench* and *AIOpsLab*.
ITBench: Jha et al. [25] proposed the ITBench framework, which adopts a holistic approach by targeting three distinct operational personas: Site Reliability Engineers (SRE), focused on availability; Chief Information Security Officers (CISO), focused on compliance and security; and Financial Operations (FinOps), focused on

cost efficiency. Consequently, ITBench aims to evaluate end-to-end incident management rather than solely SRE-related tasks. In this architecture, agents interact with the environment via specialized toolboxes that translate natural language requests into executable API commands (e.g., *NL2Kubectl* for Kubernetes operations and *NL2Metrics* for Prometheus queries).

AIOpsLab: In contrast, Chen et al. [26] introduced AIOpsLab, a framework specifically tailored to assess autonomous agents acting as Site Reliability Engineers. The conceptual foundation was previously described by Shetty et al. (2024) [27] in a vision paper detailing the design principles. AIOpsLab supports various fault scenarios and evaluates distinct tasks: *detection* (identifying anomalies), *localization* (pinpointing affected resources), *fault analysis* (classifying fault types like misconfigurations or saturation), and *mitigation* (autonomous remediation). Interaction occurs via the Agent-Cloud Interface (ACI), a middleware that requires agents to invoke structured function calls rather than natural language commands.

For this research, we selected **AIOpsLab** as the foundational platform, leveraging its robust cluster initialization and fault injection capabilities. However, significant modifications were implemented to overcome specific limitations:

- **Interface replacement:** We replaced the native ACI with a custom MCP Server. The original ACI was found to be restrictive for extending tool capabilities, whereas MCP offers a standardized and extensible protocol for tool execution.
- **Enhanced evaluation pipeline:** The evaluation methodology was revamped to assess RCA, detection, and localization concurrently within a single experimental run.
- **LLM-as-a-Judge:** Unlike the original framework, which relied on rigid keyword matching limited to fault categories, we integrated an LLM-as-a-Judge approach, which allows for a semantic evaluation of the agent's reasoning and diagnosis.

3.2. State of the Art in AIOps

Interest in applying Artificial Intelligence to Site Reliability Engineering has grown rapidly in recent years, with contemporary research predominantly focusing on LLM-based approaches that use the LLM as a central reasoning engine while implementing architectural safeguards to mitigate limitations. Several works have focused on automating incident resolution through multi-agent collaboration: Zhang et al. [28] presented *FLASH*, a workflow automation agent incorporating status supervision; Chen et al. [29] introduced *STRATUS*, which emphasizes mitigation safety via a *Transactional Non-Regression* strategy; and Pei et al. [30] proposed a SOP-Enhanced Multi-Agent System that uses standard operating procedures. A common drawback of these systems is their reliance on custom integration layers, which hinders the adoption of new observability platforms and often leads to hallucinated resource names. In contrast, our proposed approach utilizes the Model Context Protocol (MCP) as a connectivity layer to ensure scalability and validate resource existence. Beyond pure agentic workflows, Xiang et al. [31] presented *SynergyRCA*, which employs a graph-based method to feed the LLM with resource relationships; however, maintaining consistent state in this manner is computationally expensive, prompting our work to adopt a *hybrid approach* where topology is decoupled from dynamic data retrieval. Finally, distinct from these generative approaches, statistical methods like *CausalRCA* (Xin et al.) [32] utilize causal inference for fault localization but lack the semantic understanding of logs provided by modern LLM-based systems.

4. Proposed Approach

This section outlines the proposed approach. We first introduce the high-level architecture (Section 4.1) of the multi-agent system, followed by a detailed explanation of how cluster knowledge is represented (Section 4.2) and how agents communicate with external tools and one another (Section 4.3). Subsequently, we describe the multi-agent workflow (Section 4.4), defining the tasks and structure of each agent, and finally, we present the evaluation protocol (Section 4.5).

4.1. High-Level Architecture

The proposed **MAS** (Figure 1) employs four distinct agents, each specializing in specific tasks. This approach utilizes the LLM as a reasoning engine capable of analyzing significantly more raw data than a human operator, thereby providing a holistic view of the system. Decomposing the process across multiple agents divides the RCA into discrete steps; this mitigates the risk of hallucination and ensures that each node specializes in its assigned function. First, a hybrid deterministic-generative agent performs triage by checking resource status and compiling a list of symptoms. These are passed to the Planner, which generates a list of RCA tasks

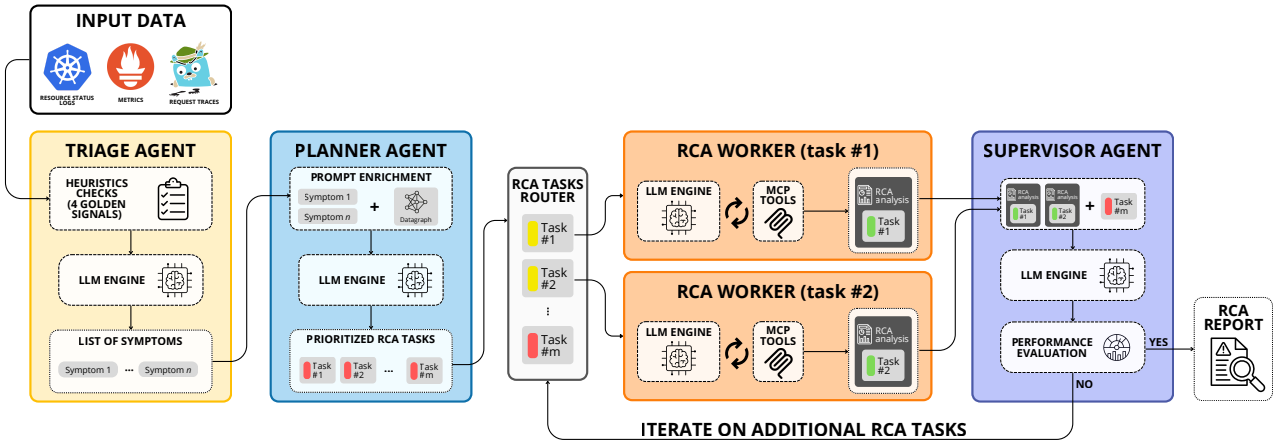


Figure 1: **Architecture of the proposed MAS.** The pipeline begins with the *Triage Agent* (yellow), which identifies symptoms from telemetry data. The *Planner Agent* (blue) enriches these symptoms to generate prioritized investigation tasks. These tasks are orchestrated by the *RCA Router*, a deterministic component that dispatches tasks to parallel *RCA Workers* (orange) and manages pending requests. Finally, the *Supervisor Agent* (purple) aggregates results and decides whether to finalize the report or trigger a feedback loop to schedule remaining tasks.

by considering the symptoms alongside the cluster topology (i.e., dependencies and upstream services). To reduce Time To Resolution, a divide-and-conquer strategy is implemented: as soon as the Planner defines the tasks, multiple RCA Workers are spawned in parallel to execute simultaneous analyses. Finally, a Supervisor agent aggregates the results from the workers to formulate the final diagnosis. If the gathered information is insufficient, the Supervisor can assign additional tasks to the RCA Workers.

4.2. Knowledge Representation

Providing agents with distilled information is essential to minimize the risk of hallucination and reduce token consumption; consequently, the method of information storage is paramount. Memory optimization is performed at two distinct levels: the agent state, which governs how information gathered throughout the multi-agent workflow is stored and shared across agents, and the Datagraph, a graph which represent the cluster topology. The latter is fundamental for guiding the LLM during cluster exploration, preventing the analysis of irrelevant resources and the consequent waste of tokens.

4.2.1 Agent's State

The agent's state is implemented as a **typed dictionary**. This structure forces the agent to store only essential information in a consistent format, avoiding reliance on unstructured strings which are often verbose and increase token usage. The multi-agent workflow utilizes a common shared state containing all necessary information accessible to subsequent agents. Additionally, each of the four agents maintains a specific local state comprising the shared data alongside internal data exclusive to that agent.

4.2.2 Datagraph

The *datagraph* is a **graph-based representation** (Figure 2) of the Kubernetes **cluster topology**. It details the connections among services, specifically categorizing them into: *data dependencies*, defined as the upstream services required by a component to perform its task; and *infrastructure dependencies*, which encompass critical backing services such as databases and caches that are essential for the service's operation.
While the **datagraph** represents services statically defined in Kubernetes manifests, Pods are dynamic entities; multiple instances may exist simultaneously due to scaling policies or ReplicaSets. Consequently, it is necessary to provide the agents with a mapping between services and their corresponding Pods, ensuring that the specific resources involved in the analysis are identified.
The datagraph is constructed using two primary data sources. Data dependencies are identified using Jaeger [33], a distributed tracing tool designed to monitor and debug microservice architectures by tracking service interactions. Infrastructure dependencies are discovered through the direct inspection of cluster Pods (e.g., by

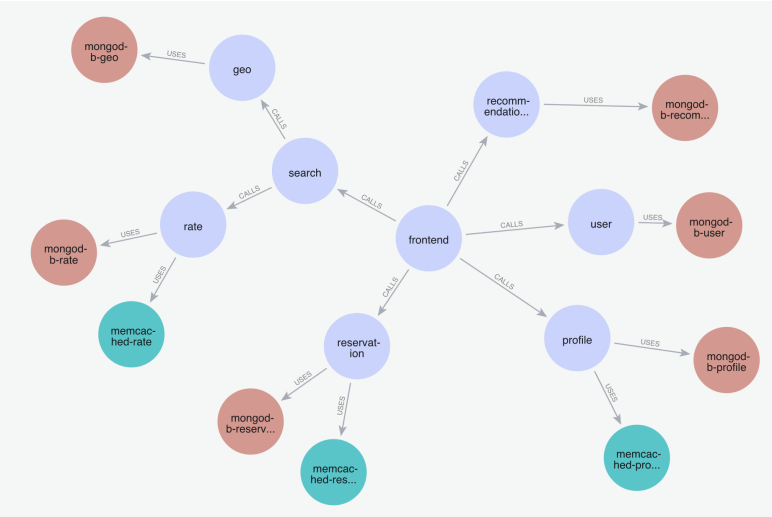


Figure 2: **Datagraph of the Hotel Reservation testbed application.** Node colors represent resource types: light blue for services, teal for caches, and red for databases. Additionally, there are two relationship types: `USES` for infrastructure dependencies and `CALLS` for data dependencies

analyzing Pod names and labels). While other methods could be employed to derive a similar topology, it is crucial to distill this information, ensuring the agent receives only the relevant data necessary for reasoning.

4.3. Communication Layer

To enable communication between agents and cluster resources, the **MCP** is employed. This approach standardizes communication and provides a suite of tools compatible with heterogeneous systems. As a standard protocol, MCP offers out-of-the-box support for authentication and security best practices, such as Human-in-the-Loop mechanisms. Furthermore, adopting MCP servers facilitates the integration of community-developed tools; consequently, incorporating additional features such as a new monitoring service is significantly more streamlined compared to developing a custom communication protocol.
To optimize context window usage by distilling the information provided to the agent, a **custom MCP server** is implemented. This server offers a suite of tools to interact with the monitoring layer, including the Kubernetes CLI, a log viewer, Prometheus for metrics gathering, and Jaeger for trace tracking. The primary objective is to **distill and digest the data** before returning it to the agent. This approach minimizes potential naming errors during tool invocation and reduces the risk of LLM hallucinations caused by the ingestion of excessive raw data.
The MCP server implements the following core features:

- **Log retrieval:** This tool enables the extraction of logs from a Pod with filtering capabilities to isolate relevant data. A keyword-based matching filter is implemented to return only rows containing specific severity levels, such as `WARN`, `ERR`, or `FATAL`, which are critical for RCA. This filtering mechanism decreases the volume of text returned to the agent, thereby reducing token consumption and minimizing the risk of LLM hallucination.
- **Trace retrieval:** This tool allows the agent to gather data regarding traffic traces within the microservice architecture. The retrieval process follows a two-tiered approach. First, the agent obtains an overview: a condensed view containing the service call path, execution time, error flags, and trace ID. This overview can be filtered to show only traces containing errors or those exceeding a latency threshold. Subsequently, the agent can request trace details by specifying the ID of a trace to be inspected. This strategy enables the agent to request only necessary information, significantly reducing token usage by digesting the data and optimizing the context window.
- **Metric retrieval:** Implemented via the Prometheus [34] API, this tool is crucial for preventing metric naming errors. Previous works have shown that agents often fail to predict the correct metric names when invoking monitoring tools. To address this, the proposed tool accepts a Pod name and returns a pre-defined set of relevant metrics (e.g., CPU usage, packet loss percentage). This ensures the agent possesses all necessary metrics for the resource throughout the analysis, eliminating redundant queries and wasted tokens caused by failed tool calls. Furthermore, the tool supports historical data retrieval; by specifying a time range, it returns an array of values for each metric.
- **Datagraph interaction:** A set of tools designed to query the Datagraph. These enable the agent to

identify upstream services and dependencies for any given component. Additionally, they facilitate the retrieval of all Pods associated with a specific Service and, conversely, the Service associated with a given Pod.

For every tool described above, the mapping between Services and Pods is preserved. The agent can invoke a tool using either a Pod name or a Service name; the MCP server processes the request to return the specific Pod information or the list of Pods associated with the given Service. This abstraction further mitigates potential naming errors during tool invocation.

4.4. Multi-Agent Workflow

The proposed approach relies on a MAS designed to automate the diagnosis of faults within the Kubernetes cluster. As illustrated in Figure 1, the workflow orchestrates four specialized agents (Triage, Planner, RCA Workers, and Supervisor) alongside a deterministic routing component.

4.4.1 Triage Agent

The Triage Agent serves as the entry point of the multi-agent workflow. It employs a hybrid approach, combining deterministic heuristics with LLM-based reasoning. Initially, a set of deterministic checks is performed to detect potential symptoms. The raw data obtained from these checks is then passed to the LLM engine, which structures the output into a defined object specifying the affected resource name, resource type, symptom description, and supporting evidence. This strategy mitigates agent hallucinations and reduces false positives by grounding the initial detection in heuristic verification. However, implementing an exhaustive set of checks is crucial to ensure no faults remain undetected.

The set of checks is derived from the Four Golden Signals, a framework established by Google's SRE team to comprehensively monitor distributed system health. A specific check is introduced for each signal:

- **Latency:** Represents the time required to service a request. It is mapped by collecting traces with execution times exceeding a defined threshold.
- **Traffic:** Measures the demand placed on the system. In this implementation, traffic is not explicitly mapped because the monitoring system lacks application-level throughput metrics (e.g., via sidecar proxies), and the available benchmarks only provide container-level metrics.
- **Errors:** Represents the rate of failed requests. This is mapped via two checks: first, by identifying Pods in problematic states (e.g., frequent restarts), and second, by analyzing network errors derived from Pod metrics.
- **Saturation:** Measures the extent to which a service approaches its resource capacity limits. This is mapped by monitoring thread saturation and CPU load metrics gathered from the Pods.

A fallback mechanism is implemented to address cases where heuristic checks yield no results. In such instances, the agent receives an overview of traces containing error flags. Although filtering for error flags mitigates false negatives, processing trace data introduces noise, as the LLM must analyze unstructured raw data, potentially leading to misleading symptoms.

Finally, the results of these checks are passed to the LLM, which generates a list of decoupled symptoms to be forwarded to the subsequent agent, the Planner.

4.4.2 Planner Agent

The Planner Agent receives the list of symptoms identified by the Triage Agent as input. Its primary role is to define an RCA strategy to gather the necessary information for the analysis. The Planner generates a prioritized list of tasks; for each task, it explicitly defines the sub-analysis goal, the target resource to investigate, and a list of suggested tools.

Decomposing the main RCA process into smaller sub-tasks is crucial for two reasons. First, by narrowing the scope of the investigation, the agent is not required to handle the full problem context simultaneously. This focused approach reduces the risk of hallucinations caused by context window saturation. Second, this decomposition enables the **parallel execution of RCA Worker** agents (Section 4.4.4), allowing multiple tasks defined by the Planner to be processed concurrently, thereby significantly reducing the Time to Resolution (TTR).

The Planner defines the task list by analyzing the reported symptoms within the context of the network topology provided by the Datagraph. For each affected resource identified by the Triage Agent, a list of upstream services and dependencies is provided in a structured format. This ensures the agent has comprehensive information regarding the involved services, enabling it to plan checks on critical dependencies such as databases or caches. This mechanism introduces a **topology-aware capability** to the agent, preventing the scheduling of unnecessary analyses on resources unrelated to the affected component.

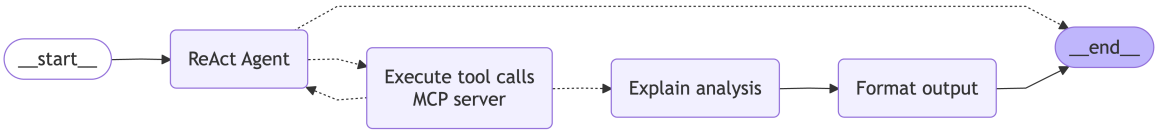


Figure 3: **Internal execution workflow of the RCA Worker agent.** The diagram illustrates the implementation of the ReAct pattern, characterized by an iterative loop between the reasoning engine (*ReAct Agent*) and the tool execution layer (*Execute tool calls MCP server*). Upon satisfying the investigation criteria or exhausting the budget, the flow transitions to the *Explain analysis* node to synthesize the interaction history for human interpretability, followed by the *Format output* node to structure the final diagnosis for the Supervisor agent.

Finally, the comprehensive list of available tools, along with their descriptions, is passed to the agent via the system prompt. This is accompanied by a set of operational rules that strictly enforce the analysis of resources topologically connected to the affected component.

4.4.3 RCA Tasks Router

The RCA Router is a **deterministic component** positioned between the Planner and the RCA Workers. Its main role is to receive the prioritized task list, manage the status of each task (assigning pending or in progress states), and initialize the RCA Workers. In the first iteration, the router dispatches the top-P tasks with the highest priority. In subsequent steps, when the execution is driven by the Supervisor Agent, the router assigns the requested pending tasks. It also acts as a filter to prevent redundancy, ensuring that the same task is not executed more than once.

4.4.4 RCA Worker Agent

Once the Planner Agent defines the task list, the P tasks with the highest priority are sent to RCA Worker agents for parallel execution by the RCA Router. The RCA Worker acts as the **execution node**, performing the specific analysis defined in the assigned task and submitting the results to the Supervisor. This workflow implements a **divide and conquer paradigm**: the Planner decomposes the RCA process into narrowed sub-tasks (Divide), each RCA Worker executes a specific analysis (Conquer), and the Supervisor finally aggregates the individual results to formulate the global diagnosis (Combine).
The RCA Worker (figure 3) is implemented using the **ReAct pattern**. This involves a continuous loop between the agent and the available tools: at each iteration, the agent invokes one or more tools, waits for the output, and subsequently determines the next action based on the observation. The complete chat history between the agent and the tools is appended to every LLM call to maintain the context. The agent accesses monitoring tools, specifically Jaeger, Prometheus, the Datagraph, and the Kubernetes CLI, via the MCP server. For safety reasons, tools capable of modifying the cluster state are disabled since the objective is strictly limited to Root Cause Analysis rather than automated mitigation. Furthermore, the agent is permitted to invoke multiple tools within a single iteration, thus reducing the total number of iterations and the overall TTR.
An alternative variant of the ReAct pattern was evaluated before adopting the standard approach described above. This variant included an intermediate summarization node designed to condense tool outputs to save tokens. However, this method was discarded because the summarization process often discarded critical context, preventing the agent from converging on a solution and causing it to repetitively invoke the same tools.
Given the importance of token efficiency, a **tool call budget mechanism** is implemented alongside task decomposition. This strategy balances context retention with token usage. By decomposing the problem, the iterations required by a single RCA Worker are significantly fewer than those required for an end-to-end analysis. The budget mechanism strictly enforces a limit on the number of tool calls per task. The remaining budget is injected into the system prompt at each iteration; if the budget is exceeded, a custom logic forces the agent to call the submission tool immediately. This constraint makes the agent to prioritize relevant actions and prevents infinite loops.
Once the analysis is completed, each RCA Worker submits a report, specifying the diagnosis and the underlying reasoning. Additionally, a separate LLM call processes the full chat history to populate a structured object

with insights and a summary of steps taken, thereby enhancing human explainability. To manage the execution flow, the state of each task includes a status field:

- *Pending:* The task has been generated by the Planner but not yet assigned.
- *In Progress:* The task is currently being executed by an RCA Worker.
- *Completed:* The RCA Worker has submitted the final report.

This state tracking ensures that tasks are executed correctly and prevents redundant processing of the same investigation.
All analyses produced by the RCA Workers are stored in the shared state. Once the high-priority tasks are completed, the list of analyses is forwarded to the Supervisor.

4.4.5 Supervisor Agent

Once all in-progress RCA analyses are submitted by the RCA Workers, the Supervisor Agent initiates the **aggregation process** to synthesize the results and formulate the final Root Cause Analysis report. To achieve this, the Supervisor receives three distinct inputs: the complete list of symptoms identified by the Triage Agent, the structured analyses submitted by the RCA Workers for the executed tasks, and the list of pending tasks originally defined by the Planner but not yet executed due to lower priority.
The primary objective of the Supervisor is to evaluate these inputs and determine whether to finalize the report or to request the execution of pending tasks to gather additional data. This introduces a **feedback loop** mechanism that enhances the system's robustness, ensuring that the diagnosis is not constrained solely to the initial set of high-priority tasks. However, this **iterative approach** presents a trade-off: if the agent requests the execution of further tasks, both token usage and the Time to Resolution (TTR) will increase, as this necessitates spawning additional RCA Workers and incurring further LLM calls.
If the Supervisor determines that sufficient evidence exists, it generates the final report. This document contains a description of the identified root cause, a list of resources affected by the incident, a synthesis of the evidence discovered by the RCA Workers, and a summary of the investigation process. Furthermore, to facilitate evaluation, two specific fields are included: the name of the faulty resource (Localization) and a boolean flag indicating whether an anomaly was confirmed (Detection).

4.5. Evaluation Protocol

The performance of the proposed agent is evaluated across three primary dimensions. First, the **accuracy** of the analysis; second, the **LLM cost**, which is evaluated to assess the economic efficiency of the system based on total token consumption and the pricing of the selected model; and third, the **Time to Resolution (TTR)**, defined as the total execution time of the multi-agent system from the initialization of the Triage Agent to the final output of the Supervisor Agent.
To assess the accuracy and quality of the models, three specific measures are employed:

- **Detection Accuracy:** the percentage of experiments in which the agents correctly detect the presence of an anomaly.
- **Localization Accuracy:** the frequency with which the system correctly identifies the root cause resource. This is evaluated using *substring matching*: since the ground truth is typically defined at the Service level (static), while the agent often identifies specific Pods (whose names are dynamic and include a unique suffix), substring matching ensures that a Pod identification is correctly mapped to its parent Service.
- **RCA Score:** to evaluate the semantic quality of the analysis, the *LLM-as-a-Judge* [35] technique is employed. A high-capability LLM compares the RCA summary generated by the multi-agent system against the ground truth description of the fault scenario. The Judge provides a score on a scale of 1 to 5, accompanied by a rationale that enhances human explainability.

While detection and localization accuracy align with the standard metrics established by the AIOpsLab [26] framework, this work augments the evaluation protocol with the **RCA Score**. This addition addresses the need to evaluate the qualitative content of the final report, going beyond the binary success metrics used in previous works.

5. Experimental Setup

This section details the experimental setup. We begin by introducing the cluster infrastructure (Section 5.1) and the integration with AIOpsLab (Section 5.2), which serves as a platform to deploy testbeds and inject faults. Subsequently, we describe the suite of testbeds and the fault types (Section 5.3). The discussion then moves to the system configuration, detailing the specific agent configurations (Section 5.4) used for the comparison, the selected LLM models (Section 5.5) and the monitoring infrastructure (Section 5.6). Finally, we outline the

automated evaluation pipeline (Section 5.7) used to assess agent performance across all experiments and the data processing performed (Section 5.8).

5.1. Cluster Setup

The Kubernetes cluster is instantiated using KinD [36] (Kubernetes in Docker), a tool designed to run **local Kubernetes clusters** within Docker containers. We use a custom configuration utilizing a specific image provided by AIOpsLab to deploy the control-plane and worker node. This configuration exposes several ports to facilitate access to the APIs of monitoring tools, such as Jaeger and Prometheus.
Furthermore, to mitigate DockerHub download rate limits and optimize deployment times given the frequent image pulls, a local Docker registry is deployed within a dedicated container to act as a cache. Alongside the cluster, an instance of Neo4j [37], a graph database, is installed to host the Datagraph for the testbed.
The entire infrastructure is hosted on an Ubuntu 24.04.3 LTS instance running on a Proxmox Virtual Machine (VM) equipped with 32GB of RAM, 16 CPU cores, and 256GB of storage. It is worth noting that the virtualization layer occasionally impacted cluster performance; specifically, disk I/O latency affected the responsiveness of etcd, the distributed key-value store used by Kubernetes for cluster state management.

5.2. AIOpsLab Integration

The AIOpsLab framework is utilized exclusively for testbed deployment, fault injection, and workload generation. In contrast, the Multi-Agent System and its communication with the cluster are fully orchestrated using MCP and LangGraph [38].
Regarding evaluation, while the AIOpsLab codebase is leveraged to extract ground truth data, specifically the faulty service for localization and the description of the injected fault, the evaluation pipeline itself is custom-built. This approach facilitates a more granular assessment of the multi-agent system's output and enables the collection of extensive runtime telemetry using specialized observability tools for LLM applications, such as LangSmith.

5.3. Testbed Applications

The AIOpsLab evaluation framework provides a suite of **three testbed applications**, defined by Helm charts for deploying the microservice networks. Each configuration includes a monitoring layer, offering built-in access to Pod metrics via Prometheus and request tracing via Jaeger. Additionally, the suite includes a workload generator designed to simulate real-world traffic scenarios.
Although the AIOpsLab scripts are generally compatible with KinD, specific modifications were required to fully support the proposed multi-agent system. First, the workload script was adjusted to extend the traffic generation duration, ensuring sufficient data availability over time. Second, the API endpoints of the monitoring tools were exposed to make them accessible to the custom MCP server running outside the cluster.
The three testbed applications available in the suite are:

- **Hotel Reservation:** part of the DeathStarBench [39] suite for cloud microservices. It implements a hotel reservation system composed of 8 microservices alongside their infrastructure dependencies, such as caches and databases.
- **Social Network:** also part of the DeathStarBench suite, this application implements a social network featuring unidirectional follow relationships. It is composed of 14 microservices plus their backing services, communicating primarily via Remote Procedure Calls (RPCs). This testbed presents a higher complexity compared to Hotel Reservation, while utilizing the same monitoring layer.
- **Astronomy Shop:** created by the OpenTelemetry [40] team, this project aims to demonstrate standard telemetry data collection. It implements an e-commerce shop featuring the same observability layer as the previous testbeds, with the addition of application-level metrics exposed via the OpenTelemetry SDK. However, this distributed service is highly resource-intensive; due to the disk I/O latency limitations mentioned in section 5.1, it was not possible to run experiments on this testbed, as the virtualization caused repeated *etcd* crashes and deployment failures.

Each testbed application includes a diverse set of fault types, ranging from *misconfigurations* (e.g., incorrect port mappings) and *network faults* (e.g., request delays or packet drops) to *scalability issues* (e.g., replica reduction or Pod termination without recreation). Fault injection mechanisms vary depending on the specific fault and application, though the open-source Chaos Mesh [41] library is predominantly employed.
It is worth noting that due to the virtualized Kubernetes environment (KinD), certain faults did not function as described in the original AIOpsLab documentation. Consequently, a total of 17 distinct fault types across two testbed applications (Hotel Reservation and Social Network) were selected to evaluate the multi-agent system.

5.4. Agent Configurations

While the underlying architectural skeleton of the Multi-Agent System remains invariant across all experiments, the system's behavior can be significantly modulated by customizing specific parameters and system prompts. The following properties define a configuration:

- **Initial Parallelism (P):** Represents the number of RCA tasks forwarded to the Workers in parallel during the **first iteration** (immediately following the Planner's analysis). Specifically, the system executes the **top-P prioritized tasks concurrently**. It is worth noting that this parameter is directly proportional to computational cost, as spawning multiple simultaneous agents increases the aggregate token consumption.
- **Tool Call Budget (B):** Defines the maximum number of allowed tool invocations an RCA Worker can perform before being forced to submit a solution.
- **System Prompts (S):** Each agent operates based on a **specific system prompt** that dictates its behavior, constraints, and step-by-step instructions. These prompts are injected into the shared state, allowing for dynamic behavior modification without code changes.

Formally, each experimental configuration is identified by a unique ID and defined by the tuple $\langle P, B, S \rangle$.

5.4.1 System Prompt Strategies

This section details the distinct prompt strategies (S) designed to evaluate different behavioral patterns.

- **Plain ReAct (tool-guided planning)** (★): The baseline configuration. No custom domain-specific constraints are applied. The Planner agent is instructed to **suggest a specific set of tools** for each RCA task, guiding the Worker's investigation path.
- **Tool-free Planning** (■): A variation of the baseline where the Planner defines the investigation goal but strictly **abstains from suggesting specific tools**. This configuration grants the **RCA Worker full autonomy** in selecting the most appropriate tools to resolve the assigned task.
- **Improved ReAct** (♠): Designed for complex, stateful applications (e.g., the Social Network testbed). The Planner is explicitly instructed to **analyze the topological connections between services**, infrastructure dependencies, and Kubernetes-specific resources (e.g., Secrets, ConfigMaps). Furthermore, while the Planner suggests tools, the RCA Worker is authorized to deviate from these suggestions if runtime observations indicate a more relevant investigative path.
- **Conservative ReAct** (♣): A **token-efficient strategy** that enforces brevity and minimizes hallucination. It instructs the Triage agent to report only symptoms supported by hard evidence and enforces the RCA Worker to avoid verbose reasoning. The final report is required to be strictly concise.
- **Strict Supervisor** (▲): This configuration targets the Supervisor agent, enforcing a high threshold for certainty. The Supervisor is instructed to **prioritize iterative validation** over speculation, rejecting inconclusive reports. This often triggers a feedback loop in which pending tasks are scheduled for execution based on the evidence gathered in previous steps.

A total of **twelve distinct agent** configurations, derived from combinations of these parameters and prompt strategies, were tested across all fault scenarios. These configurations are detailed in Table 1.

System Prompt (S)	$P = 2$	$P = 3$	$P = 5$
Plain ReAct (★)	**A**	**B**	**C**
Conservative (♣)	**D**	**E**	–
Tool-free (■)	**F**	**G**	–
Strict Supervisor (▲)	**H**	**I**	–
Improved ReAct (♠)	**J**, **L***	**K**	–

* Configuration L has $B = 15$

Table 1: **Agent configurations.** All configurations utilize a Budget $B = 7$, with the exception of configuration **L** where $B = 15$.

5.5. LLM model

The LLM employed as the reasoning engine for the proposed MAS is **GPT-5-mini** [42, 43], a compact OpenAI model with a knowledge cutoff of May 31, 2024. In contrast, the AIOpsLab baseline relies on **GPT-4** [44],

which has been traditionally adopted as a flagship, high-capacity model in prior works. This setup intentionally reflects a **flagship-versus-compact** comparison rather than a strict like-for-like model substitution.
The choice of GPT-5-mini is primarily motivated by **practical deployment considerations** common in real-world AIOps scenarios, namely **cost-efficiency**, **scalability**, and **context capacity**. GPT-5-mini provides a substantially larger context window (up to 400,000 tokens) compared to the standard GPT-4 configuration (8,192 tokens), which is particularly beneficial for ReAct-based agentic workflows where interaction histories grow rapidly and may otherwise exceed context limits.
Recent public benchmarks [45] indicate that GPT-5-mini can outperform GPT-4 on selected reasoning-oriented evaluations. In particular, GPQA [46] focuses on **graduate-level physics question answering**, covering domains such as quantum mechanics, statistical physics, and classical mechanics, and is designed to assess isolated scientific reasoning in a single-turn question-answering setting. While these results highlight strong analytical capabilities, GPQA does not capture key aspects of agentic systems, such as **multi-step coordination**, **long-horizon reasoning**, or **closed-loop interactions with the environment**, which are central to the MAS setting considered in this work.
In this context, GPT-5-mini is particularly well suited for **instruction-following and well-scoped reasoning tasks**, especially when responsibilities are decomposed across specialized agents. Accordingly, the proposed system emphasizes **architectural efficiency** and **coordination mechanisms** over reliance on a single large, monolithic model, reflecting a realistic trade-off between performance and deployability in operational AIOps environments.
Finally, experimental feasibility was further enhanced by leveraging OpenAI's *"Share inputs and outputs with OpenAI"* program, enabling access to a daily quota of 2.5 million free completion tokens and allowing extensive experimentation under realistic cost constraints.
For evaluation, **GPT-5** [43, 47] is employed as an LLM-as-a-judge due to its superior reasoning and comparative assessment capabilities.

5.6. Monitoring Infrastructure

To assess the performance of the Multi-Agent System, distinct mechanisms are employed to gather runtime data and collect evaluation metrics. For each experimental run, a JSON artifact is generated, capturing the entire state of the multi-agent system alongside all metrics required for evaluation.
To measure token consumption and execution latency for each agent, the system integrates LangSmith [48], a monitoring platform designed for LangGraph applications. LangSmith provides complete visibility into agent behavior through tracing and real-time monitoring, enabling the granular tracking of performance metrics for every individual node and the persistence of these results for subsequent analysis.
Furthermore, to assess behavioral patterns, a specific analysis is performed within the *format-output* node at the conclusion of each RCA Worker's execution. The complete chat history is analyzed to quantify the frequency of tool invocations, mapping thus the distribution of tool usage across different fault scenarios.
Finally, to facilitate human explainability, a visualization interface is provided. Designed as a static web application, this tool parses the generated JSON artifacts to present the analysis results, reasoning steps, and system logs in a human-readable layout.

5.7. Automated Evaluation Pipeline

Evaluating the MAS requires a complex sequence of operations: cluster provisioning, testbed deployment, fault injection, traffic generation, and agent execution. To ensure reproducibility and manage the combinatorial complexity of testing multiple agents across various fault scenarios, an Automated evaluation pipeline has been developed. The high-level logic of this pipeline is formalized in Algorithm 1.
The pipeline operates by iterating over two primary sets of configuration files, stored as JSON artifacts:

- **Fault Scenarios:** These files define the **environmental specifications**, including the testbed application to deploy, the API endpoints for monitoring services, the associated Datagraph, and the ground truth data required for evaluation.
- **Agent Configurations:** These files determine the **agent's runtime behavior**, specifying system prompts, parallelism settings for RCA tasks, and tool call budgets. Additionally, a `runs` key allows defining the number of independent executions per configuration to ensure statistical relevance.

Both configuration types support an `execute` boolean flag, enabling the selective inclusion or exclusion of specific scenarios or agents from the batch run.
Upon initialization, the pipeline parses these configurations and presents an execution plan, calculating the total number of required runs. Following user confirmation, the iterative process begins. To guarantee experimental isolation and prevent state contamination, the Kubernetes cluster is deleted and reprovisioned via AIOpsLab

CLI whenever the fault scenario changes.
During execution, the pipeline handles the orchestration of the Multi-Agent System and the subsequent data gathering from LangSmith. A safety mechanism utilizing the OpenAI API checks the token usage before each run to prevent exceeding daily quotas. Finally, a notification system (integrated via Telegram) monitors the execution status, alerting the user to critical errors or batch completion.

Algorithm 1 Automated SRE Agent Evaluation Pipeline

```
scenarios ← LoadFaultScenarios()
agentConfigs ← LoadAgentConfigurations()
batchDir ← GetBatchDirectory()
for each scenario in scenarios do
    if TokenUsageExceeded() then
        Abort execution
    end if
    UpdateDatagraph(scenario.name)
    SetupClusterAndFault(scenario.AIOpsLabCommand)
    for each config in agentConfigs do
        if TokenUsageExceeded() then
            Abort execution
        end if
        ApplyConfigOverrides(config)
        Wait(scenario.waitTime)
        ⟨result, execTime⟩ ← RunSREAgent(scenario, config)
        executionData ← GatherExecutionData(result, execTime, config)
        evaluation ← EvaluateAgainstGroundTruth(executionData, scenario)
        SaveResults(executionData, evaluation, batchDir)
    end for
    CleanupCluster()
end for
```

5.8. Data Processing

Upon the completion of experimental batches, a dedicated data processing pipeline is employed to **aggregate the collected metrics** and **visualize the results** through tables and plots, grouped by agent configuration, testbed application, and fault type.
This pipeline supports the aggregation of data across different batches of experiments. Given the iterative nature of the agent development, a specific **deduplication policy** is applied: when merging datasets, the system identifies unique experiment identifiers (pairing agent configuration and fault scenario) and prioritizes the most recent execution based on timestamps.
Furthermore, to validate the reliability of the observed performance differences, particularly concerning token usage and execution time, a statistical analysis module is utilized. This module performs hypothesis testing by comparing repeated executions (e.g., $N = 30$ runs) of different agent configurations on identical fault scenarios. This approach allows for the assessment of the **statistical significance** of the results, distinguishing meaningful improvements from stochastic variance inherent in LLM-based systems.

6. Experimental Results and Discussion

In this section, we present and discuss the experimental findings obtained from the evaluation campaign. We begin by analyzing the impact of the core architectural components (Section 6.1) and the efficiency of the proposed tooling mechanism (Section 6.2). Subsequently, we dissect the system's performance across distinct fault typologies to derive insights into specific failure modes (Section 6.3). The discussion then proceeds to a comprehensive comparative analysis of the tested Agent Configurations (Section 6.4), supported by a statistical hypothesis test to validate the efficiency gains of the Planner-guided approach with respect to the Tool-free (Section 6.5). Finally, we benchmark our results against the current State of the Art to evaluate the performance of the proposed Multi-Agent System (Section 6.6).

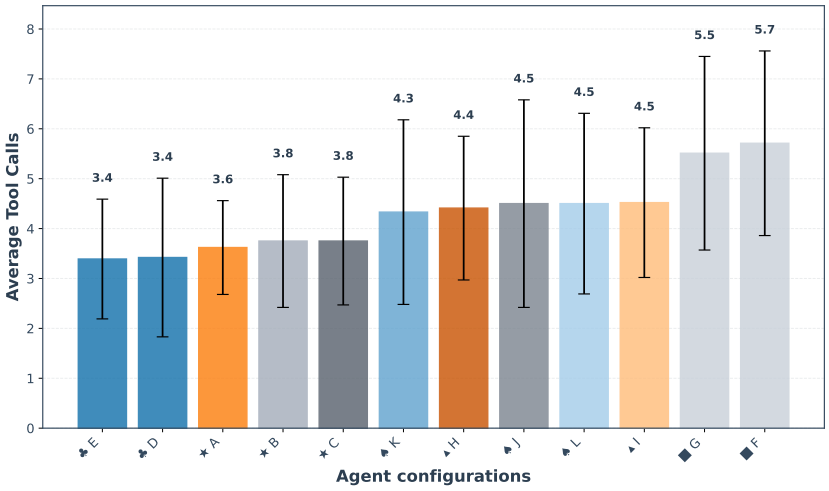


Figure 4: **Average number of tool calls performed by the RCA Worker per iteration.** The data suggests that configurations employing the *Plain ReAct* strategy (A, B) require significantly fewer tool invocations (approx. 3.6–3.8) compared to the *Tool-free Planning* configurations (F, G), which exhibit the highest activity (approx. 5.5–5.7). The increased autonomy in F and G, where the Planner provides no tool suggestions, necessitates broader exploration by the Worker, resulting in higher token consumption and extended execution latency.

Experimental protocol All experiments are executed through the automated evaluation pipeline described in Section 5.7, and results are aggregated as described in Section 5.8.

6.1. Impact of Architectural Components

The proposed architecture integrates specialized components designed to enhance both diagnostic accuracy and operational efficiency. First, the **Triage Agent** employs a deterministic approach that resolves the "false positive" problem. By relying on the aggregation of heuristic checks rather than open-ended exploration, it achieved an accuracy of **100%** in the "No Fault" detection task (i.e., scenarios where no faults were injected), as reported in the *No fault* row of Table 2 (Det.). Furthermore, the implementation of a fallback logic, which analyzes raw error traces when standard heuristics yield no signals, proved critical for identifying subtle failure modes, such as Network faults, which often do not trigger explicit metric saturation.
The **Planner Agent**'s decomposition strategy proved critical for optimizing resources. As illustrated in Figure 4, configurations where the Planner explicitly suggests tools (Plain ReAct, ★) require significantly **fewer tool calls** on average compared to tool-free (■) configurations. This suggests that scoping the analysis to topologically connected resources prevents the RCA Worker from performing broad, expensive exploratory actions. However, this reliance on the Planner introduces a trade-off: it acts as a **single point of failure**, where an incorrect plan caused by an inaccurate topology can cascade into a failed analysis.
Finally, the parallelism in the **RCA Worker** phase offers dual benefits. Beyond reducing execution time, it mitigates the risk of **hallucination**: by scoping each worker to a specific sub-task, the context window remains focused and is not saturated by the irrelevant history of the entire investigation. Experimental results indicate that finding an exact optimal number of initial tasks is not strictly necessary; spawning just two tasks ($P = 2$) is sufficient to achieve high performance (as seen in configurations D and F - Table 3), largely because the **Supervisor Agent** can dynamically schedule remaining pending tasks if the initial evidence proves inconclusive.

6.2. Tooling and Protocol Efficiency

The implementation of a custom MCP server constrains the agent's action space more effectively than relying only on prompt engineering. Furthermore, the MCP server **enforces validation logic** that prevents operations on non-existent resources. These features drastically reduce tool invocation errors, effectively blocking the LLM from hallucinating invalid resource names or incorrect metric queries. Consequently, this contributes to a **reduction in the average number of tool calls** per RCA Worker (Figure 4), as the agent does not waste ReAct iterations correcting or handling error feedback. On the other hand, while the verbosity of the Model Context Protocol increases the input token usage, this computational cost is justified by the significant increase

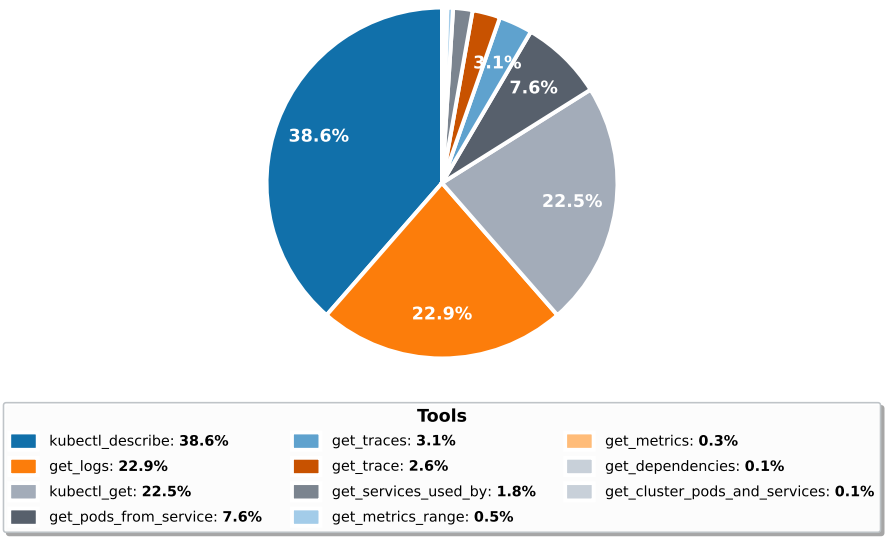


Figure 5: **Distribution of tool calls during RCA tasks.** The usage pattern highlights a reliance on textual and structural data. While descriptive tools like `kubectl_describe` and logs dominate, the agent also actively utilizes topology-related tools (e.g., `get_pods_from_service`, $\approx 7.6\%$) to map dependencies. Conversely, numerical metric retrieval remains negligible ($< 1\%$), confirming that the model prioritizes semantic context and structural awareness over raw time-series analysis.

in robustness and the reduction in execution errors.
Regarding the specific tool usage patterns observed within the RCA Worker's ReAct loop, experimental analysis (aggregated across all RCA tasks in the full evaluation campaign) revealed several distinct behavioral strategies (see Figure 5 for the empirical distribution of tool calls):

- The agent consistently adopts a **"broad-to-narrow" investigation strategy**. It begins by seeking a high-level overview of the system state before narrowing the scope to pinpoint specific anomalies. A prime example is the interaction with tracing tools: instead of immediately requesting voluminous raw data, the agent first invokes `get_traces`, which returns a lightweight summary list including error flags and trace paths. Based on this output, the agent selectively invokes `get_trace` in subsequent iterations to fetch the full raw content only for the traces identified as relevant. This approach significantly minimizes context saturation (and consequently, the risk of hallucination) while optimizing total token consumption by avoiding the retrieval of irrelevant data.
- Experimental data reveals a marked **preference for textual data sources**, such as logs, Kubernetes object descriptions (like `kubectl describe`), and request traces, over the raw numerical metrics exposed by Prometheus (`get_metrics`). This bias does not arise from interface usability issues, as the metric retrieval tool correctly exposes valid metric names, preventing hallucination. Instead, this behavior likely reflects a limitation of Large Language Models, which are optimized for processing explicit semantic signals found in text rather than interpreting raw numerical time-series data.
- Finally, the agent extensively **utilizes topology interrogation tools** to map the relationships between Pods and Services. By explicitly querying for both data and infrastructure dependencies connected to the affected resource, the agent effectively grounds its analysis in the actual cluster state, facilitating the precise identification of misconfigurations and network connectivity issues.

6.3. Fault Specific Insights

The Multi-Agent System was evaluated across two distinct testbed applications, covering a total of 17 specific fault types. The corresponding *per-fault* averages for detection (Det.), localization (Loc.), execution time, token usage, cost, and RCA score are reported in Table 2. Since Table 2 aggregates results by fault type (and does not explicitly separate the two testbeds), the comparison between *Hotel Reservation* and *Social Network* is discussed qualitatively: in our runs, the *Hotel Reservation* benchmark was generally easier to diagnose, primarily due to its lower architectural complexity and more explicit microservice interactions. Although the observability stack was identical for both environments, the Social Network proved more challenging; we hypothesize that a more granular exposure of application-level metrics such as request rates and specific failure rates would significantly

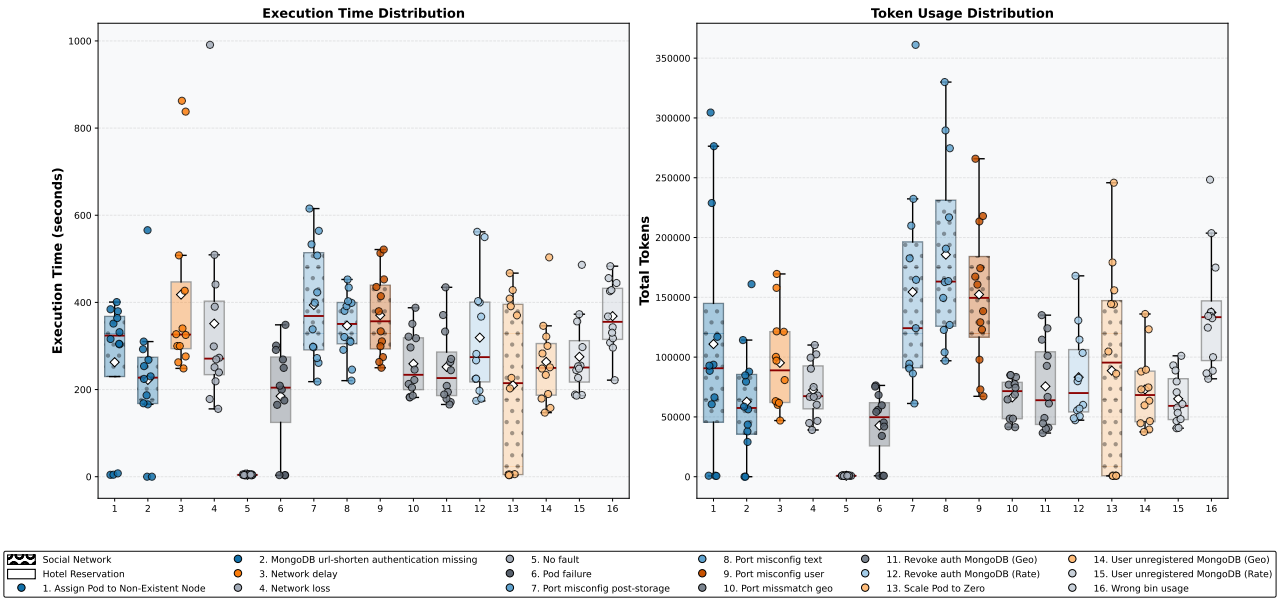


Figure 6: **Performance distribution by fault type across testbeds.** The boxplots indicate significant variability in execution time and token usage for specific fault scenarios. Notably, faults such as *Pod Failure* (6) and *Scale Pod to Zero* (13) exhibit extreme outliers or broader distributions. These anomalies often correspond to experimental runs where the fault disrupted the workload generator's ability to send traffic, leading to inconsistent diagnostic signals.

enhance fault detection in such complex topologies. This improvement would allow the agent to complement log analysis with high-value quantitative signals, which could also be leveraged by specialized models.

As detailed in Table 2, the agent achieved its highest performance on the "Port mismatch Geo" fault within the Hotel Reservation testbed. In this scenario involving a MongoDB port misconfiguration triggering a *CrashLoopBackOff* in the Geo pods, the diagnosis was facilitated by the high informational content of the logs and the topological awareness provided by the Datagraph. The graph explicitly highlighted the infrastructure dependency on the database, directing the agent to verify the communication channel. Broader analysis confirms that the agent excels in scenarios where log analysis provides sufficient evidence, such as configuration mismatches or authentication failures, while traces and metrics are used mainly to support the evidences rather than primary diagnostic signals.

Conversely, performance degrades when the diagnosis necessitates deep analysis of raw trace data. As previously discussed, while LLMs struggle with numerical time-series due to architectural limitations, the challenge with distributed traces is distinct: processing a full Jaeger trace (typically a large JSON object) consumes a vast number of tokens while offering low information density. This results in a **"garbage in, garbage out" phenomenon** where the model, overwhelmed by the **high noise-to-signal ratio** of the raw data, struggles to extract sparse relevant details and consequently begins to hallucinate, leading to imprecise analysis. This limitation is quantitatively reflected in Table 2: for example, the log-driven *Port mismatch geo* fault reaches Det.=1.00, Loc.=1.00, and RCA=4.25, whereas network faults such as *Network delay* achieve only Loc.=0.58 and RCA=1.08 (Det.=1.00), despite a much higher average runtime (417.9s) and token usage (95,207). This indicates that, in the absence of aggregated application-level metrics, the agent is often forced to parse raw traces, resulting in poorer alignment with the ground truth.

Finally, the stability of the workload generator emerged as a distinct experimental bottleneck. As visible in the variance shown in Figure 6 and Figure 7, certain fault types exhibit extreme performance fluctuations. This occurs primarily when the injected fault severs the application's entry point: if the workload generator cannot complete requests, the cluster generates zero telemetry (logs or metrics), making the fault effectively invisible to the agent. Although we attempted to mitigate this by increasing request volume and adjusting the script logic, severe infrastructure faults still resulted in sporadic detection gaps. For instance, *Pod failure* reaches only Det.=0.75 and Loc.=0.75, while *Scale Pod to Zero* drops to Det.=0.58 and Loc.=0.25 (Table 2), despite comparable execution times (184.9s and 209.8s, respectively). This confirms that in AIOps evaluation, the ability of the testbed to maintain traffic flow during failure conditions is a critical variable.

Fault Name	Det.	Loc.	Time (s)	Tokens	Cost ($)	RCA
Port mismatch geo	1.00	1.00	259.2	66,330	0.03	4.25
No fault	1.00	1.00	4.5	786	0.00	3.62
Revoke auth MongoDB (Rate)	1.00	1.00	319.2	83,185	0.04	3.58
User unreg. MongoDB (Geo)	1.00	1.00	263.4	72,989	0.03	3.17
Revoke auth MongoDB (Geo)	1.00	1.00	252.0	75,564	0.03	2.50
MongoDB url-shorten auth missing	1.00	0.92	222.2	62,732	0.03	1.58
Network loss	1.00	0.92	351.3	72,462	0.04	1.00
User unreg. MongoDB (Rate)	1.00	0.83	275.2	65,120	0.03	3.00
Pod failure	0.75	0.75	184.9	42,790	0.02	1.00
Port misconfig text	1.00	0.67	346.7	185,687	0.05	2.17
Network delay	1.00	0.58	417.9	95,207	0.05	1.08
Port misconfig user	1.00	0.42	370.7	152,353	0.05	1.67
Assign Pod to Non-Existent Node	0.75	0.42	262.8	110,911	0.03	1.17
Scale Pod to Zero	0.58	0.25	209.8	88,670	0.03	1.25
Port misconfig post-storage	1.00	0.17	394.0	205,015	0.05	2.33
Wrong bin usage	1.00	0.08	368.5	137,274	0.05	1.00

Note: Det. = Detection Acc.; Loc. = Localization Acc.; Tokens = Avg. Total Tokens; Cost = Avg. LLM cost per run ($); RCA = Avg. RCA Score.

Table 2: **Detailed experimental results by fault type.** The table highlights the disparity in performance based on the available telemetry. Explicit configuration faults, such as *Port mismatch geo*, achieve optimal detection and localization scores (1.00) with high RCA score (4.25), as the agent effectively leverages logs and topology. In contrast, scenarios reliant on raw trace analysis or lacking application-level metrics, such as *Network loss* or *Wrong bin usage*, exhibit significantly lower localization accuracy and RCA scores (1.00), reflecting the "garbage in, garbage out" challenge associated with high-noise trace data.

6.4. Comparative Analysis of Agent Configurations

A core contribution of this work is the **systematic evaluation** of the Multi-Agent System across varying configurations. By modifying parameters such as the system prompt, the number of initial parallel tasks (P), and the tool call budget (B) within the same architectural skeleton, we assessed a total of 12 distinct configurations (as detailed in Section 5.4).

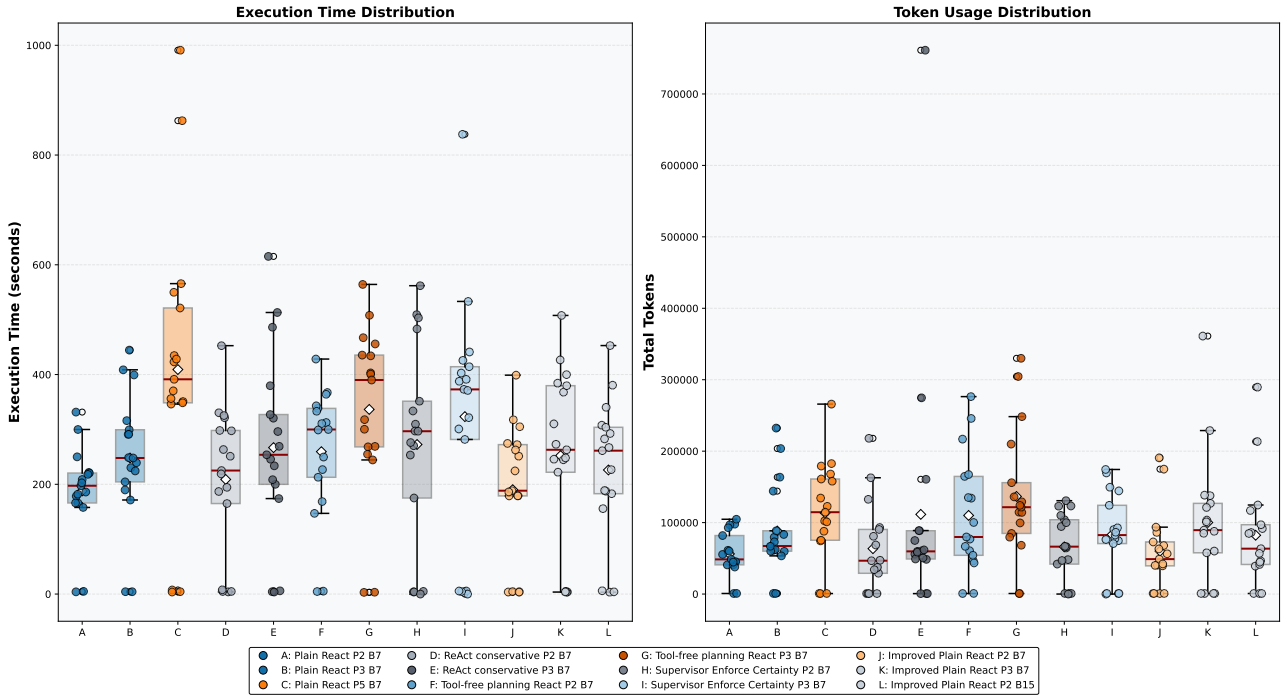
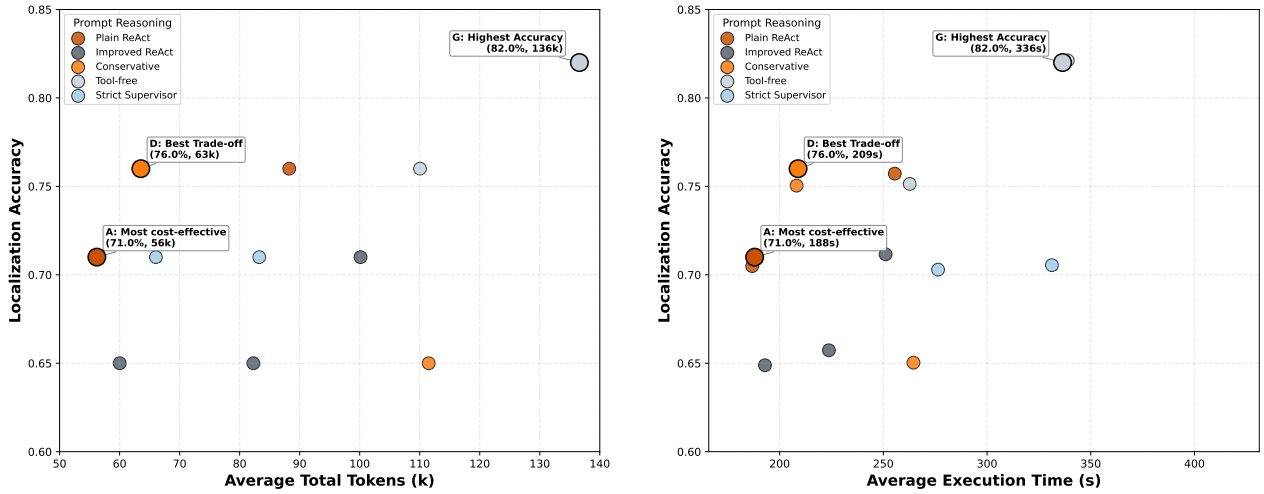


Figure 7: **Resource consumption distribution by Agent Configuration.** The boxplots show the spread of execution time and token usage for each agent variant. The evident lower outliers (near 0) across all groups represent successful "Early Termination" events triggered by the Triage module during *No Fault* scenarios or instances of insufficient telemetry generation. This confirms the architecture's ability to minimize costs when deep analysis is unwarranted.



(a) **Cost Efficiency**: Accuracy vs. Token Usage

(b) **Time Efficiency**: Accuracy vs. Execution Time

Figure 8: **Efficiency vs Accuracy Trade-off.** Comparison of agent configurations based on resource consumption. **(a)** illustrates the trade-off between localization accuracy and Token Usage (Cost), while **(b)** highlights the relationship with Execution Time. Agent **D** offers the best balance for both metrics, whereas Agent **G** maximizes accuracy at the expense of higher latency and cost. Agent **A** proves to be the most cost-effective solution for resource-constrained scenarios.

As detailed in Table 3 and Figure 8, **Agent G** (tool-free planning, $P = 3, B = 7$) emerged as the **top performer** in terms of raw accuracy, achieving **100% detection** and **82% localization**. However, this performance comes at a significant computational cost: it is the most expensive configuration in terms of token usage ($\approx$ 137k tokens) and LLM cost ($\approx$ \$0.04 per run), with a relatively high execution time ($\approx$ 336s). The high resource consumption

is caused by the "tool-free" nature of the Planner; without specific tool guidance, the RCA Worker operates with broader instructions, necessitating a more exhaustive, and consequently more expensive, exploration of the action space.

In contrast, **Agent D** (Conservative ReAct, $P = 3, B = 7$) represents the **optimal balance** between performance and efficiency. By utilizing a system prompt that enforces brevity, this configuration achieved a localization accuracy of 0.76 while consuming less than half the tokens ($\approx$ 64k) and requiring significantly less time ($\approx$ 209s) compared to Agent G. Although the detection accuracy was slightly lower (0.88), likely attributable to the aforementioned workload generator instabilities, Agent D achieved the second-highest **RCA Score** (2.59). This suggests that constraining the agent's reasoning and output length forces the Supervisor to process only distilled, high-value information, thereby preventing context window saturation and reducing the likelihood of hallucination.

Configurations **F** (tool-free, $P = 3$) and **B** (Plain ReAct, $P = 3$) also delivered strong results, acting as a middle ground. Both achieved 100% detection accuracy with 76% localization (Table 3, Det./Loc.). Notably, **Agent B** (Plain ReAct) proved more efficient than **Agent F** (tool-free): it consumed $88,260$ tokens vs. $110,043$ tokens and required $247.9s$ vs. $259.5s$ on average (Table 3, Tokens/Time). This further validates that Planner-guided tool suggestions reduce the token overhead required for the Worker to identify the correct diagnostic actions.

Agent A (Plain ReAct, $P = 2, B = 7$) stands out as the **most cost-effective configuration**. With the lowest token usage ($\approx$ 56k) and fastest execution time (188s), it still maintained 100% detection and 71% localization accuracy. This finding is significant as it demonstrates that **massive parallelism is not always required**; a modest number of initial tasks ($P = 2$) combined with the efficient ReAct strategy is sufficient to capture the majority of faults.

Conversely, **Agent C** (Plain ReAct, $P = 5, B = 7$) yielded the **lowest localization accuracy** (0.59) despite achieving the highest RCA Score (2.76). This apparent paradox can be attributed to information overload: spawning 5 parallel RCA tasks generates a volume of content that saturates the Supervisor's context window. While the Supervisor receives enough detail to describe the incident types accurately (high RCA Score), the noise makes it difficult to pinpoint the exact affected resource (low localization), leading to hallucinations regarding the fault location.

Finally, the "Strict Supervisor" and "Improved ReAct" variants did not yield remarkable improvements, suggesting that hardcoded iterative checks or rigid communication rules are less effective than optimizing the prompt strategy. A notable finding from **Agent L** (Improved ReAct, $P = 2, B = 15$) is that even with a generous budget of 15 tool calls, the RCA Worker averaged only 4.5 calls per task (shown in Figure 4). This confirms the efficacy of the "divide and conquer" design: since the tasks have been strictly defined by the Planner, the Workers rarely need deep exploration to fulfill their specific objectives.

ID	Type	P	B	Det.	Loc.	Time (s)	Tokens	Cost ($)	RCA
G	■	3	7	1.00	0.82	336.4	136,601	0.04	2.24
D	♣	2	7	0.88	0.76	208.9	63,541	0.03	2.59
F	■	2	7	1.00	0.76	259.5	110,043	0.03	2.29
B	★	3	7	1.00	0.76	247.9	88,260	0.03	1.88
A	★	2	7	1.00	0.71	188.0	56,212	0.03	2.29
K	♠	3	7	0.88	0.71	252.3	100,135	0.03	2.29
I	▲	3	7	0.94	0.71	323.4	83,267	0.04	2.12
H	▲	2	7	0.94	0.71	272.5	66,046	0.03	1.59
L	♠	2	15	0.94	0.65	226.1	82,290	0.03	2.41
J	♠	2	7	0.88	0.65	190.2	60,019	0.02	2.29
E	♣	3	7	0.94	0.65	266.9	111,503	0.03	2.06
C	★	5	7	0.94	0.59	409.1	113,513	0.04	2.76

Note: P = Parallelism RCA; B = Tool budget; Det. = Detection Acc.; Loc. = Localization Acc.; Tokens = Avg. Total Tokens; Cost = Avg. LLM cost per run ($); RCA = Avg. RCA Score.

Table 3: **Agent performance comparison.** The table ranks agent configurations based on a hierarchical sorting strategy: first by localization accuracy (*Loc.*) in descending order, followed by the RCA Score (*RCA*) in descending order, and finally by Token usage (*Tokens*) in ascending order. The columns **P** and **B** indicate the number of initial parallel tasks and the tool call budget, respectively.

6.5. Hypothesis Testing: Efficiency of Tool-Guided Planning

To quantify the efficiency benefits of tool-guided planning (Agent A) versus the tool-free approach (Agent F), we conducted a **statistical comparison**. We specifically targeted these two configurations as they represent the most efficient implementations within their respective classes (both operating with a parallelism of $P = 2$). The evaluation consisted of **30 independent runs** across five distinct fault scenarios. For each scenario, we analyzed two metrics: *Execution Time* and *Total Token Usage*. To rigorously validate the performance gap, we formulated a **one-sided hypothesis test**:

- **Null Hypothesis ($H_0 : \mu_A \geq \mu_F$):** The tool-guided approach does not offer any efficiency gain over the tool-free approach (i.e., its resource usage is equal to or greater than that of Agent F).
- **Alternative Hypothesis ($H_1 : \mu_A < \mu_F$):** The tool-guided approach significantly reduces resource usage compared to Agent F.

Depending on the normality of the data distribution (verified via Shapiro-Wilk), we applied either the **Student's t-test** or the non-parametric **Mann-Whitney U test** with a significance level of $\alpha = 0.05$.

The results, detailed in Table 4, provide strong statistical evidence supporting the proposed architecture. We **rejected the null hypothesis (H_0) in 9 out of 10 comparisons**, confirming that Agent A yields a statistically significant reduction in resource consumption. Specifically, the tool-guided planner reduced token usage by up to **42%** and execution time by approximately **29%**. The only exception was the execution time for the *SN - Auth Missing* scenario ($p = 0.103$), where we failed to reject H_0, indicating that the time difference in this specific low-latency case was not statistically significant.

Scenario	Metric	Mean A	Mean F	Imp.	p-val	Test
HR - Port mismatch geo	Time (s)	196.5	275.3	28.6%	**0.000**	MW-U
	Tokens	41,663	71,821	42.0%	**0.000**	MW-U
HR - User unreg. MongoDB	Time (s)	200.0	275.8	27.5%	**0.000**	MW-U
	Tokens	44,887	74,570	39.8%	**0.000**	MW-U
SN - Port misconfig post-storage	Time (s)	310.7	382.4	18.7%	**0.001**	MW-U
	Tokens	141,495	232,757	39.2%	**0.000**	MW-U
SN - Auth Missing	Time (s)	217.9	225.8	3.5%	0.103	MW-U
	Tokens	47,219	63,780	26.0%	**0.000**	MW-U
HR - Revoke auth MongoDB	Time (s)	196.1	268.1	26.9%	**0.000**	T-Test
	Tokens	42,752	72,732	41.2%	**0.000**	MW-U

Table 4: Statistical comparison between Agent A and Agent F (One-sided hypothesis test $H_1 : A < F$). The prefixes **HR** and **SN** denote the *Hotel Reservation* and *Social Network* scenarios, respectively. Significant results ($p < 0.05$), where H_0 is rejected, are marked in bold.

6.6. Benchmarking Against State-of-the-Art

To validate the efficacy of our approach, we compared our system against *AIOpsLab* [26], a framework that evaluates single-agent architectures, including standard ReAct patterns, Microsoft's FLASH [28], and GPT-4 wrappers, across detection, localization, and RCA tasks.

It is important to note that a direct comparison involves certain methodological asymmetries. First, due to virtualization constraints in our experimental environment, we could not deploy the "Astronomy Shop" scenario; consequently, our aggregated results are derived from the *Hotel Reservation* and *Social Network* benchmarks, whereas AIOpsLab's reported data represent an aggregation across their full test suite. However, given the significant performance margin observed, we consider the comparison to be indicative of the architectural advantages. Second, a direct comparison of computational overhead (execution time and token usage) is infeasible due to fundamental architectural differences. Unlike our unified end-to-end execution, AIOpsLab treats each task as a discrete experiment (requiring three separate runs for a full diagnosis). Furthermore, our adoption of a MAS utilizing the verbose MCP naturally incurs a higher token overhead than their single-agent, custom-interface approach. Consequently, we focus our comparative analysis on **accuracy metrics**, specifically *detection accuracy* and *localization accuracy (@1)*. We exclude the RCA metric from this comparison, as AIOpsLab employs a categorical classification approach, whereas our system utilizes an LLM-as-a-Judge for a semantic evaluation of the root cause description.

As shown in Table 5, we benchmark our top-performing configuration, **Agent G** (tool-free planning, $P = 3, B = 7$), and our most balanced configuration, **Agent D** (Conservative ReAct, $P = 3, B = 7$), against AIOpsLab's best-performing agents. In the **detection** task, our Agent G matches the state-of-the-art performance of FLASH (100%), while the efficiency-focused Agent D achieves 88%. In the **localization** task, our system demonstrates significant superiority: while AIOpsLab's best agent (GPT-4-W-SHELL) achieves an accuracy of 61.54%, our Agent G reaches 82% (+20.46 percentage points, i.e., +33.25% relative), and even the resource-efficient Agent D achieves 76% (+14.46 percentage points, i.e., +23.50% relative).

These results were achieved using *GPT-5-mini*, a cost-effective model optimized for specific agentic tasks. In contrast, AIOpsLab relies on the flagship GPT-4, a general-purpose model that is approximately 120x more expensive for input processing and 30x for output processing. The fact that our Multi-Agent Architecture matches or outperforms a monolithic baseline using a smaller and cheaper model validates our hypothesis: effective task decomposition is a stronger determinant of RCA success than raw model parameter count.

Task	Approach	Agent/Model	Accuracy	Gain
Detection	SoTA (AIOpsLab)	FLASH	100.00%	-
	Ours (Best)	**Agent G (GPT-5-mini)**	**100.00%**	=
	Ours (Trade-off)	Agent D (GPT-5-mini)	88.00%	-12.00%
Localization	SoTA (AIOpsLab)	GPT-4-W-SHELL (GPT-4)	61.54%	-
	Ours (Best)	**Agent G (GPT-5-mini)**	**82.00%**	**+20.46%**
	Ours (Trade-off)	Agent D (GPT-5-mini)	76.00%	+14.46%

Table 5: **Comparison with State-of-the-Art (AIOpsLab).** Accuracy comparison between the best AIOpsLab single-agent baselines (using GPT-4) and our Multi-Agent System configurations (using GPT-5-mini).

7. Conclusions

This thesis aimed to advance the exploration of **MAS** for RCA in Kubernetes environments. By implementing a Divide-and-Conquer design alongside specific strategies to mitigate the "AI in the Wild" problem, such as a custom MCP for data distillation, heuristic-based Triage, and topological grounding via a Datagraph, we demonstrated that RCA can be effectively performed using compact, cost-efficient models specialized for narrow tasks, rather than relying on expensive, general-purpose flagship models.

Throughout the analysis, several key architectural findings emerged. First, the **hybrid triage approach** eliminated false positives; by using heuristics to digest raw data, the LLM is triggered only when actual anomalies are detected, thereby decoupling symptom detection from reasoning. Second, the adoption of the **datagraph** proved crucial for the planning phase, enabling the Planner Agent to map the dependencies of the affected resource and construct a targeted investigation plan. Third, the custom **MCP server** played a vital role by enforcing strict invocation schemas and providing distilled telemetry, which significantly reduced tool-calling errors and context saturation. Finally, the **divide and conquer** strategy allowed for the decomposition of complex RCA workflows into smaller sub-tasks executed in parallel, reducing the overall time to resolution.

A significant contribution of this work was the comparative analysis of different agent configurations to evaluate the influence of parameters and prompting strategies. We discovered that a **tool-guided approach**, where the Planner Agent explicitly prescribes the tools for each task, is substantially more efficient than a tool-free approach. This guidance prevents the RCA Worker from performing broad, useless analyses, thereby saving tokens and execution time. Furthermore, the **"conservative"** configuration, which enforces brevity in agent responses, emerged as the optimal trade-off in terms of token usage, execution time, localization accuracy, and RCA score. This highlights the importance of maintaining a concise context window to speed up analysis and mitigate hallucinations.

Finally, we benchmarked our approach against the state-of-the-art framework, **AIOpsLab**, which represents the paradigm of using monolithic flagship LLMs for end-to-end RCA. Our Multi-Agent System improved **localization accuracy** from 61.54% (AIOpsLab) to **82%** (Our Best Agent), corresponding to a +20.46 percentage-point gain (i.e., a +33.25% relative improvement).

In conclusion, this research validates the efficacy of the Multi-Agent System approach for Root Cause Analysis. The results confirm that the raw reasoning power of the model is not the primary determinant of success; rather, it is the integration of architectural guardrails, deterministic mechanisms, and efficient task decomposition that ensures a reliable and accurate analysis.

7.1. Limitations and Future Work

Despite the promising results demonstrated by the proposed MAS, certain limitations must be acknowledged. The primary technical constraint lies in the analysis of raw distributed traces. Even with current distillation mechanisms, high-volume trace data can rapidly saturate the context window, reducing the signal-to-noise ratio and potentially obscuring critical path latencies. Furthermore, while Large Language Models excel at semantic reasoning, they exhibit inherent limitations in interpreting raw numerical time-series data. Relying solely on the LLM to detect anomalies within metric ranges can be imprecise, particularly for fault types where log data is sparse or insufficient, such as network issues.

Finally, from an experimental perspective, the validation was subject to resource constraints. Access to expanded computational resources would enable performing tests over the third testbed application, *Astronomy Shop*, and

covering the whole suite of fault types. Additionally, the current experiments were conducted on a virtualized Kubernetes environment (KinD). While suitable for functional testing, a virtualized network stack may not perfectly emulate the complex noise found in large-scale, bare-metal production clusters.
To address these limitations, future iterations of this work will focus on integrating specialized Deep Learning models, such as autoencoders, alongside the LLM. In this setup, the specific model would act as a precise detector for time-series anomalies, passing only a high-level "anomaly signal" to the agent, thereby offloading the numerical interpretation from the language model. Additionally, building on the success of using smaller models like GPT-5-mini, the next logical step is to shift towards distilled Small Language Models (SLMs) hosted directly within the cluster. By fine-tuning these models for specific sub-tasks such as log parsing or plan generation, the entire RCA pipeline could run locally. This would not only eliminate third-party API costs but also guarantee that sensitive operational data never leaves the infrastructure. Future validation will also move beyond virtualization to deploy the system on real multi-node clusters, testing the agents' robustness against the unpredictability of physical hardware.

References

[1] Benjamin H Sigelman et al. "Dapper, a Large-Scale Distributed Systems Tracing Infrastructure". en. In: (). URL: https://research.google/pubs/dapper-a-large-scale-distributed-systems-tracing-infrastructure/.

[2] Chris Jones, Betsy Beyer, Niall Richard Murphy, and Jennifer Petoff. *Site Reliability Engineering: How Google Runs Production Systems*. en. 2016. ISBN: 1-4919-2912-X. URL: https://sre.google/sre-book/.

[3] Lingjiao Chen, Matei Zaharia, and James Zou. *FrugalGPT: How to Use Large Language Models While Reducing Cost and Improving Performance*. arXiv:2305.05176 [cs]. May 2023. DOI: 10.48550/arXiv.2305.05176. URL: http://arxiv.org/abs/2305.05176.

[4] Nelson F. Liu et al. "Lost in the Middle: How Language Models Use Long Contexts". In: *Transactions of the Association for Computational Linguistics* 12 (Feb. 2024), pp. 157–173. ISSN: 2307-387X. DOI: 10.1162/tacl_a_00638. URL: https://doi.org/10.1162/tacl_a_00638.

[5] Ehud Karpas et al. *MRKL Systems: A modular, neuro-symbolic architecture that combines large language models, external knowledge sources and discrete reasoning*. arXiv:2205.00445 [cs]. May 2022. DOI: 10.48550/arXiv.2205.00445. URL: http://arxiv.org/abs/2205.00445.

[6] Timo Schick et al. *Toolformer: Language Models Can Teach Themselves to Use Tools*. arXiv:2302.04761 [cs]. Feb. 2023. DOI: 10.48550/arXiv.2302.04761. URL: http://arxiv.org/abs/2302.04761.

[7] Marcello Martini. *Github repository SRE-Agent*. URL: https://github.com/martinimarcello00/SRE-agent.

[8] Oluwayemisi Runsewe, Olajide Soji Osundare, Samuel Olaoluwa Folorunsho, and Lucy Anthony Akwawa. "SITE RELIABILITY ENGINEERING IN CLOUD ENVIRONMENTS: STRATEGIES FOR ENSURING HIGH AVAILABILITY AND LOW LATENCY". In: *Acta Electronica Malaysia* 8.1 (Oct. 2024), pp. 31–38. ISSN: 25904043. DOI: 10.26480/aem.01.2024.31.38. URL: https://www.actaelectronicamalaysia.com/archives/AEM/2024-issue1/1aem2024-31-38.pdf.

[9] Andrew Ifesinachi Daraojimba et al. "Systematic Review of Key Performance Metrics in Modern DevOps and Software Reliability Engineering". en. In: *International Journal of Future Engineering Innovations* 1.1 (2024), pp. 101–107. ISSN: 30491215. DOI: 10.54660/IJFEI.2024.1.1.101-107. URL: https://www.futureengineeringjournal.com/search?q=FEI-2025-1-026&search=search.

[10] Laxmi Rijal, Ricardo Colomo-Palacios, and Mary Sánchez-Gordón. "AIOps: A Multivocal Literature Review". en. In: (2022). Ed. by Sanjay Misra, Amit Kumar Tyagi, Vincenzo Piuri, and Lalit Garg. Book Title: Artificial Intelligence for Cloud and Edge Computing ISBN: 9783030808204 9783030808211 Place: Cham, pp. 31–50. DOI: 10.1007/978-3-030-80821-1_2. URL: https://link.springer.com/10.1007/978-3-030-80821-1_2.

[11] Youcef Remil, Anes Bendimerad, Romain Mathonat, and Mehdi Kaytoue. "AIOps Solutions for Incident Management: Technical Guidelines and A Comprehensive Literature Review". In: (2024). Version Number: 1. DOI: 10.48550/ARXIV.2404.01363. URL: https://arxiv.org/abs/2404.01363.

[12] Qian Cheng et al. "AI for IT Operations (AIOps) on Cloud Platforms: Reviews, Opportunities and Challenges". In: (2023). Version Number: 1. DOI: 10.48550/ARXIV.2304.04661. URL: https://arxiv.org/abs/2304.04661.

[13] Jian Hou. *Research on fault diagnosis and root cause analysis based on full stack observability*. arXiv:2509.12231 [cs]. Sept. 2025. DOI: 10.48550/arXiv.2509.12231. URL: http://arxiv.org/abs/2509.12231.

[14] Enoch Solomon, Abraham Woubie, and Bernabas Solomon Emiru. "Systematic Review of Large Language Models and Future Directions". In: *2025 8th International Conference on Information and Computer Technologies (ICICT)* (Mar. 2025). Conference Name: 2025 8th International Conference on Information and Computer Technologies (ICICT) ISBN: 9798331505189 Place: Hawaii-Hilo, HI, USA, pp. 1–6. DOI: 10.1109/ICICT64582.2025.00036. URL: https://ieeexplore.ieee.org/document/11058466/.

[15] Humza Naveed et al. "A Comprehensive Overview of Large Language Models". en. In: *ACM Transactions on Intelligent Systems and Technology* 16.5 (Oct. 2025), pp. 1–72. ISSN: 2157-6904, 2157-6912. DOI: 10.1145/3744746. URL: https://dl.acm.org/doi/10.1145/3744746.

[16] Ashish Vaswani et al. *Attention Is All You Need*. arXiv:1706.03762 [cs]. Aug. 2023. DOI: 10.48550/arXiv.1706.03762. URL: http://arxiv.org/abs/1706.03762.

[17] Alec Radford et al. "Language Models are Unsupervised Multitask Learners". en. In: ().

[18] Yubo Wang et al. *MMLU-Pro: A More Robust and Challenging Multi-Task Language Understanding Benchmark*. arXiv:2406.01574 [cs]. Nov. 2024. DOI: 10.48550/arXiv.2406.01574. URL: http://arxiv.org/abs/2406.01574.

[19] Aisha Alansari and Hamzah Luqman. *Large Language Models Hallucination: A Comprehensive Survey*. arXiv:2510.06265 [cs]. Oct. 2025. DOI: 10.48550/arXiv.2510.06265. URL: http://arxiv.org/abs/2510.06265.

[20] Wenqi Fan et al. "A Survey on RAG Meeting LLMs: Towards Retrieval-Augmented Large Language Models". In: *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. KDD '24. New York, NY, USA: Association for Computing Machinery, Aug. 2024, pp. 6491–6501. ISBN: 979-8-4007-0490-1. DOI: 10.1145/3637528.3671470. URL: https://dl.acm.org/doi/10.1145/3637528.3671470.

[21] Mohamad Abou Ali and Fadi Dornaika. "Agentic AI: A Comprehensive Survey of Architectures, Applications, and Future Directions". In: *Artificial Intelligence Review* 59.1 (Nov. 2025). arXiv:2510.25445 [cs], p. 11. ISSN: 1573-7462. DOI: 10.1007/s10462-025-11422-4. URL: http://arxiv.org/abs/2510.25445.

[22] Shunyu Yao et al. *ReAct: Synergizing Reasoning and Acting in Language Models*. arXiv:2210.03629 [cs]. Mar. 2023. DOI: 10.48550/arXiv.2210.03629. URL: http://arxiv.org/abs/2210.03629.

[23] Lei Wang et al. *Plan-and-Solve Prompting: Improving Zero-Shot Chain-of-Thought Reasoning by Large Language Models*. arXiv:2305.04091 [cs]. May 2023. DOI: 10.48550/arXiv.2305.04091. URL: http://arxiv.org/abs/2305.04091.

[24] *Model Context Protocol documentation*. URL: https://modelcontextprotocol.io/docs/getting-started/intro.

[25] Saurabh Jha et al. *ITBench: Evaluating AI Agents across Diverse Real-World IT Automation Tasks*. arXiv:2502.05352 [cs]. Feb. 2025. DOI: 10.48550/arXiv.2502.05352. URL: http://arxiv.org/abs/2502.05352.

[26] Yinfang Chen et al. *AIOpsLab: A Holistic Framework to Evaluate AI Agents for Enabling Autonomous Clouds*. arXiv:2501.06706 [cs]. Jan. 2025. DOI: 10.48550/arXiv.2501.06706. URL: http://arxiv.org/abs/2501.06706.

[27] Manish Shetty et al. "Building AI Agents for Autonomous Clouds: Challenges and Design Principles". In: *Proceedings of the 2024 ACM Symposium on Cloud Computing*. SoCC '24. New York, NY, USA: Association for Computing Machinery, Nov. 2024, pp. 99–110. ISBN: 979-8-4007-1286-9. DOI: 10.1145/3698038.3698525. URL: https://dl.acm.org/doi/10.1145/3698038.3698525.

[28] Xuchao Zhang et al. "FLASH: A Workflow Automation Agent for Diagnosing Recurring Incidents". en. In: (). URL: https://arxiv.org/html/2401.16732v2.

[29] Yinfang Chen et al. *STRATUS: A Multi-agent System for Autonomous Reliability Engineering of Modern Clouds*. arXiv:2506.02009 [cs]. May 2025. DOI: 10.48550/arXiv.2506.02009. URL: http://arxiv.org/abs/2506.02009.

[30] Changhua Pei et al. *Flow-of-Action: SOP Enhanced LLM-Based Multi-Agent System for Root Cause Analysis*. arXiv:2502.08224 [cs]. Feb. 2025. DOI: 10.48550/arXiv.2502.08224. URL: http://arxiv.org/abs/2502.08224.

[31] Yong Xiang et al. *Simplifying Root Cause Analysis in Kubernetes with StateGraph and LLM*. arXiv:2506.02490 [cs]. June 2025. DOI: 10.48550/arXiv.2506.02490. URL: http://arxiv.org/abs/2506.02490.

[32] Ruyue Xin, Peng Chen, and Zhiming Zhao. "CausalRCA: Causal inference based precise fine-grained root cause localization for microservice applications". In: *Journal of Systems and Software* 203 (Sept. 2023), p. 111724. ISSN: 0164-1212. DOI: 10.1016/j.jss.2023.111724. URL: https://www.sciencedirect.com/science/article/pii/S016412122300119X.

[33] *Jaeger: open source, distributed tracing platform*. URL: https://www.jaegertracing.io.

[34] *Prometheus*. URL: https://prometheus.io.

[35] Jiawei Gu et al. *A Survey on LLM-as-a-Judge*. arXiv:2411.15594 [cs]. Oct. 2025. DOI: 10.48550/arXiv.2411.15594. URL: http://arxiv.org/abs/2411.15594.

[36] *Kubernetes in Docker*. URL: https://kind.sigs.k8s.io/.

[37] *Neo4j*. URL: https://neo4j.com.

[38] *LangGraph*. URL: https://www.langchain.com/langgraph.

[39] Yu Gan et al. "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*. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, Apr. 2019, pp. 3–18. ISBN: 978-1-4503-6240-5. DOI: 10.1145/3297858.3304013. URL: https://dl.acm.org/doi/10.1145/3297858.3304013.

[40] *OpenTelemetry*. URL: https://opentelemetry.io.

[41] *Chaos Mesh*. URL: https://chaos-mesh.org.

[42] *GPT-5 mini*. URL: https://platform.openai.com/docs/models/gpt-5-mini.

[43] Aaditya Singh et al. *OpenAI GPT-5 System Card*. arXiv:2601.03267 [cs]. Dec. 2025. DOI: 10.48550/arXiv.2601.03267. URL: http://arxiv.org/abs/2601.03267.

[44] OpenAI et al. *GPT-4 Technical Report*. arXiv:2303.08774 [cs]. Mar. 2024. DOI: 10.48550/arXiv.2303.08774. URL: http://arxiv.org/abs/2303.08774.

[45] L. L. M. Stats. *GPT-4 vs GPT-5 mini Comparison: Benchmarks, Pricing & Performance*. en. Section: AI Model Comparisons. URL: https://llm-stats.com/models/compare/gpt-4-0613-vs-gpt-5-mini-2025-08-07.

[46] David Rein et al. *GPQA: A Graduate-Level Google-Proof Q&A Benchmark*. arXiv:2311.12022 [cs]. Nov. 2023. DOI: 10.48550/arXiv.2311.12022. URL: http://arxiv.org/abs/2311.12022.

[47] *GPT-5*. URL: https://platform.openai.com/docs/models/gpt-5.

[48] *LangSmith*. URL: https://www.langchain.com/langsmith.

Acknowledgements

Desidero ringraziare innanzitutto il Professor Marco Domenico Santambrogio per le opportunità offerte e per avermi seguito con attenzione durante gli anni della laurea magistrale. Un ringraziamento particolare va a Matteo Ferroni per la costante guida, il prezioso supporto e i consigli forniti durante l'intero sviluppo di questo progetto di tesi. Ringrazio inoltre il NECSTLab e il progetto Leonardo per avermi fornito un ambiente stimolante in cui ampliare le mie competenze pratiche e professionali.
Un ringraziamento profondo e sincero va alla mia famiglia. A mia madre Cinzia e mio padre Roberto, per il sostegno incondizionato, per i sacrifici fatti per permettermi di raggiungere questo traguardo e per aver sempre creduto in me. A mio fratello Massimo e a tutti i miei parenti, per essermi stati vicini e avermi incoraggiato nei momenti più complessi di questo percorso accademico.
Infine, il mio pensiero va ai miei amici. A quelli storici di Ferrara, punto di riferimento costante che mi accompagna da una vita; agli amici conosciuti durante gli anni della triennale, dell'Erasmus e a quelli incontrati qui al Politecnico di Milano, con i quali ho condiviso fatiche, traguardi e momenti indimenticabili. Grazie a tutti voi per il supporto e per aver reso questo viaggio speciale.