Between Promise and Pain: The Reality of Automating Failure Analysis in Microservices with LLMs

Alessandro Cornacchia
KAUST

Iliyas Alabdulaal
KAUST

Ibraheem Saghier
KAUST

Albaraa Mirdad
KAUST

Omar Fayoumi
KAUST

Marco Canini
KAUST

Abstract

Large Language Models (LLMs) are increasingly explored as general-purpose assistants for infrastructure operations, helping automate tasks like querying data, analyzing logs, and suggesting fixes. In this paper, we consider the more general and ambitious problem of fully automating root cause analysis (RCA) in microservice systems, where LLMs must collect information, reason about it, and interact with the environment to detect, localize and resolve issues. Anecdotal evidence offers useful insights and partial solutions, but the broader challenge remains unresolved. We systematically evaluate multiple LLM agent architectures across a range of incident scenarios. We study how different tool-augmented agents perform, and shed light on common failure modes, including hallucinated reasoning paths and inefficient use of context. Our findings reveal both the promise and the limitations of current approaches, and point to concrete directions for more robust and effective use of LLMs in this domain.

CCS Concepts

• **Computer systems organization** → *Cloud computing*; • **Computing methodologies** → **Intelligent agents**; • **Applied computing** → **Service-oriented architectures**.

Keywords

Cloud-native applications, observability, Large Language Models, AI agents, root cause analysis

ACM Reference Format:
Alessandro Cornacchia, Iliyas Alabdulaal, Ibraheem Saghier, Albaraa Mirdad, Omar Fayoumi, and Marco Canini. 2025. Between Promise and Pain: The Reality of Automating Failure Analysis in Microservices with LLMs. In *16th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '25), October 12–13, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3725783.3764388

APSys '25, Seoul, Republic of Korea

ACM ISBN 979-8-4007-1572-3/25/10
https://doi.org/10.1145/3725783.3764388

1 Introduction

It is notoriously difficult to diagnose issues in microservice systems, as Root Cause Analysis (RCA) [18, 20, 29, 33, 53, 57, 60] requires correlating and finding causality relations across diverse observability data and components [16, 17, 47, 59, 62]—a process that overwhelms rule-based tools, imposing significant manual burden on already strained Site Reliability Engineers (SREs). Paradoxically, despite the wide range of observability tools available today, organizations face crippling complexity to resolve incidents and minimize downtimes [30]. This complexity increases costs and can outpace the capacity of available human expertise.

Large Language Models (LLMs)–e.g., OpenAI's GPT [37], Anthropic's Claude [8], Meta's Llama [36] and several others—have demonstrated the ability to process multimodal data and engage in natural-language-driven reasoning [56]. Among the diverse application areas, LLMs can excel in several tedious subtasks that SREs routinely perform, such as crafting API invocations [2], querying metrics [61], reasoning about time-series [34, 55], logs [44] and distributed traces [7]. As a consequence, the community is naturally gravitating toward exploring their use to automate SREs' operations at large [24, 26, 42, 51, 58], with commercial cloud vendors already integrating LLM-driven tools [40, 48].

However, while anecdotal evidence has so far demonstrated some benefits of LLMs in supporting SRE subtasks, how to fully automate the end-to-end workflow of RCA and incident mitigation in microservice applications remains an open problem. For example, RCAgent [51] tackles RCA automation in the cloud, but—beyond the lack of publicly available implementations and associated private datasets—it doesn't directly target microservice applications. Similarly, AIOpsLab[45] is the first unified platform to evaluate AI agents for microservice diagnosis. However, it remains centered on experiment orchestration and benchmarking, and only surfaces the performance of LLM-based agents for end-to-end incident resolution. Therefore, it is still hard to draw general conclusions about the effectiveness of LLMs in microservice's RCA workflows.

In this paper, we share our initial experience with applying LLMs to fully automate RCA tasks in microservice applications. We implement multiple tool-augmented agent architectures and characterize their behavior in across diverse failure scenarios. More specifically, we examine and systematically catalog *the good* – e.g., successful incident localizations and effective tool usage – *the bad* – e.g., hallucinated API calls and parameter errors – and *the ugly* – e.g., largely missed objectives or undesired system actions, disruptive for the application. These findings reveal that while LLMs show promise for certain diagnostic subtasks, their decision-making processes remain prone to failure modes even when triaging apparently simple failure cases. At their core, we find that these issues can manifest even with recent reasoning models [27] and regardless of the use of advanced prompting strategy [28, 52, 56], and multi-agent architectures [32, 54]. Based on these observations, we explore potential opportunities, such as offloading telemetry processing to specialized *expert* agents – which can help improve diagnostic accuracy and tooling flexibility. Lastly, we preliminary curate a benchmark of Q&A to assess the ability of LLMs to retrieve and reason on artifacts of microservices applications, like code and documentation. Our work aims to share the current reality of automated LLM-based RCA, highlighting both capabilities and fundamental limitations and inform future research directions.

2 Related work

The application of LLM to cloud incidents is an active and rapidly evolving research area in the software engineering communities. Benchmarks and platforms to standardize the evaluation of LLM agents for DevOps are gaining popularity [10, 23, 35, 41, 45, 46]. In parallel, several works propose LLM-based approaches to anomaly detection in the cloud [5, 14, 15, 22, 34]. These are mostly coupled to subtasks like logs and time-series analysis, or assistants to navigate static troubleshooting-guides [31], but do not address automation of the entire RCA worflow. The works [4, 11, 12, 25, 26, 58] apply LLM-based methods to RCA tasks. Some assume expensive fine-tuning [4, 26], others, such as MonitorAssistant [58] only apply to private corporate settings and do not consider environment interactions. RCAgent [51] is the closest to the vision of a fully autonomous RCA agent. However, it does not directly target microservices applications. To the best of our knowledge, how to automate reasoning, decision-making, information collection and environment interactions altogether for RCA of microservices applications still remain a largely open problem.

3 Methodology

In this section, we overview the experimental setup and the LLM-based agents we implement.

Environment. We experiment with AIOpsLab [45], an open-source benchmarking platform designed to evaluate LLM agents in automated incident response scenarios. AIOpsLab provides a suite of Kubernetes-based [3] microservices applications and an extensible benchmark containing ∼40 failure cases. For each failure case, agents are prompted to determine whether an issue exists and, if so, to identify its root cause. LLM agents can interact with the microservice application and with the environment by accessing external *tools*. For example, AIOpsLab includes basic tools to run shell commands – e.g., access `kubectl` CLI – or retrieve telemetry from common observability monitors – e.g., distributed traces and metrics backends[1, 9]. We cover more details in § 4.1.2.

We make the following refinements. AIOpsLab does not natively support sequential or asynchronous fault injections in the same run, limiting the evaluation of agents in scenarios involving fault escalation, such as temporally correlated or compounding failures, or transient faults that disappear while diagnosis is still ongoing. Therefore, we incorporate an open-loop scheduling mechanism in the fault injector to control synthetic failures with more flexibility. In addition, we add new performance contentions scenarios to the failure benchmark – e.g., CPU throttling and memory leaks – to target also non-crash failure modes. We deploy the SocialNetwork and HotelReservation applications.[1]

LLM agents. We implement and evaluate multiple LLM-based agent configurations, each with a different architecture and prompting strategy (cf. Tab. 1). They vary in terms of structure (single vs multi-agent), prompting strategy (ReAct vs static prompts) and language model. First, we implement two single-agent architectures based on the ReAct framework [56], labeled `RE1` and `RE2`. ReAct is a prompting method that extends Chain-of-Thought (CoT) [52] and guides a model to alternate between the CoT's reasoning steps and external actions (e.g., interaction with environment described in § 3). We evaluate `RE1` and `RE2` using `GPT-o1` and `GPT-4o`, a reasoning and a non-reasoning model, respectively [27, 38]. Second, we explore configurations `AG1` and `AG2`, both following a collaborative multi-agent paradigm implemented using AutoGen GroupChat [54]. In this case, multiple agents communicate through message passing within a group chat-like structure, where a coordinator agent built into AutoGen manages message routing and decides who speaks next based on the conversation context.

Type	Models	Multi-agent	Ref.
AutoGen	3x GPT-4o	✓	AG1
	1x GPT-o1 + 2x GPT-4o	✓	AG2
ReAct	GPT-4o	×	RE1
	GPT-o1	×	RE2

Table 1: Agents in this paper.

[1] we deploy the implementations provided with AIOpsLab codebase [13]; the applications were first introduced in DeathStarBench [19].

We assign distinct roles to three agents: a *reasoner*, an *executor*, and a *critic*. The reasoner formulates hypothesis and instructs the executor. The executor manages interactions with the environment. The critic monitors the reasoning and execution, raising objections when inconsistencies or errors are detected. All agents have access to the complete chat history whenever invoked. For `AG1`, we use the `GPT-4o` model for all agents, while `AG2` uses `GPT-o1` for the reasoner and `GPT-4o` for the executor and critic. In all configurations, we instruct agents with an intentionally uninformative prompt with semantics: *"Check if this microservice application has any issues. If so, list three suspected root-cause services ranked by confidence in decreasing order. Then produce a summary to explain your choice."*

4 LLM agents in action: lessons learned

We describe what we learned from applying LLM agents to real diagnostic workflows. We organize our findings into successes (§ 4.1), limitations (§ 4.2), and failure modes (§ 4.3).

4.1 The Good

4.1.1 Can LLMs reason about microservices? Retrieving and reasoning on artifacts, in addition to telemetry data, is crucial for diagnostics in microservice applications. For example, identifying root causes often requires deeper code analysis, such as understanding invocation patterns or asynchronous call orders [63], which distributed traces alone cannot reveal.

Step zero is to determine whether LLMs can reason about microservice applications at all. To this end, we curate MSA-CausalBench, a set of 112 open-ended questions on the DSB Hotel application. We compare three retrieval configurations: (1) *Docs* – using natural language descriptions of the code files as context (crawling online code repositories [6, 19, 45]), (2) *Code* – using raw source code files as context, and (3) *Docs & Code* – combining natural language descriptions and source code. The questions range from assessing basic aspects of distributed applications to evaluating the model's ability to reason about service and operational dependencies. Additional details are provided in Appendix B.2.

We implement a RAG pipeline using ColBERTv2 [43], a retrieval model that enables fine-grained, matching between queries and documents at token level. We use Claude 3.5 Sonnet, which excels in code generation.

Category	Docs	Code	Both
System Infrastructure	0.88	0.94	0.94
Service Dependencies	1.00	1.00	0.94
Latency Propagation	0.63	0.88	0.63
Operational Dependencies	0.61	0.94	0.97
Request Flow	0.19	0.88	0.81
Overall	**0.59**	**0.93**	**0.89**

Table 3: Accuracy of `ColBERTv2-RAG` on `MSACausalBench` across three retrieval configurations with Claude 3.5 Sonnet.

As shown in Tab. 3, the *Code* configuration achieved the highest overall accuracy of 0.93, outperforming both *Docs* (0.59) and *Docs & Code* (0.89). This counter-intuitive result suggests that natural language descriptions might not help the agent and introduce ambiguity. The combined approach requires the LLM to reconcile redundant representations, which, for the HotelReservation codebase, reflects into lower accuracy.

Therefore, selecting what information to retrieve in a RAG setting is not as trivial and largely task-dependent.

4.1.2 Tooling integration and its benefits. Next, we turn to actual RCA and unleash the agents on the AIOpsLab benchmark to observe their behavior in realistic diagnostic problems. We compare two scenarios. In the first scenario, called *baseline*, agents only have access to the standard AIOpsLab tools, which include basic shell commands and telemetry retrieval functions like `get_logs`, `get_metrics`, and `get_traces`. In the second scenario, we augment AIOpsLab with a suite of custom tools that allow agents to query summarized observability data rather than raw telemetry.

We develop the tools listed in Tab. 2, covering a common set of observability data types, including metrics, traces, and logs. The tools can be broadly categorized into conventional non-LLM tools, such as statistical and machine learning methods, and LLM-based "expert" agents, denoted as *Agent-as-a-Tool* (AaaT). While the former are designed to provide aggregate insights (e.g., statistical summaries, clustering) to be used for rule-based detection and analysis, the latter enable more flexible conversational interactions between the diagnostic agent and expert agents to explore telemetry data.

We posit this structure offers two key advantages. First, it is neither realistic nor practical to assume that SREs would provision agents with the perfect tool for every diagnostic task. In extreme cases, this would undermine the purpose of using LLMs for automation. Modern language models can interface with execution environments, execute code dynamically, maintain variables, and revise prior actions on-the-fly [49, 50]. AaaT naturally aligns with these capabilities, enabling expert agents to compose and execute the appropriate tool for the task at hand, rather than being constrained by a predefined toolset. Second, AaaT facilitates *model specialization*, allowing fine-tuned models, such as ChatTS [55] for time-series data or Parallax [7] for distributed traces, to focus on analyzing specific types of telemetry data.

RCA experiments. We run the agents on the AIOpsLab benchmark, repeating each agent configuration 5 times. We evaluate the following dimensions. (1) *Accuracy*, where `Acc@3` measures whether the correct root-cause microservice appears in the top-3 candidates, while `Acc@1` whether the root-cause coincides with the agent's first choice. (2) *Reasoning*

Tool	Data	Description	Type
`time_series_expert`	M	Analyzes Prometheus data and returns structured textual insights (e.g., trends, anomalies).	Agent-as-a-Tool
`traces_expert`	T	Analyzes distributed traces to identify bottlenecks and critical service delays.	Agent-as-a-Tool
`time_series_summary`	M	Summarizes a target metric over a time window: count, mean, std, min/max, quartiles.	Statistical
`traces_summary`	T	Computes aggregate statistics across all traces: volume, top operations (APIs), error rates, etc.	Statistical
`api_latency_distribution`	T	Computes the probability distribution of the execution latency for different APIs.	Statistical
`cluster_time_series`	M	Groups metrics with similar behaviors using a K-Shape clustering on time series [39].	ML
`get_cluster_representatives`	M	Selects representative metrics within each cluster by using correlation and Granger causality [21, 47].	Statistical
`analyze_trace_spans`	T	Extrapolates a span-level summary given an individual trace, e.g., delay attribution.	Utility
`logs_to_error_summary`	L	Extracts timestamps of error occurrences (pattern matching) and generates reports.	Utility

Table 2: Our domain-specific tools to aid LLM agents with processing multi-modal observability data. Legend: [M]: metrics [T]: traces [L]: logs. We integrate them in AIOpsLab [45] platform.

Ref.	Acc@1	Acc@3	Time (s)	Tokens In	Tokens Out	Steps	Cost ($)
AG1	20.0	30.0	62.1	16814	1381	14.4	0.12610
	41.7 ↑	70.0 ↑	158.3	2197	3610	16.2	0.02248 ↓
AG2	70.0	70.0	236.5	39233	188	17.1	0.78465
	100.0 ↑	100.0 ↑	136.6	4867	228	9.7	0.10334 ↓
RE1	50.0	50.0	51.8	41231	1170	11.7	0.10308
	65.0 ↑	90.0 ↑	60.0	3647	601	6.9	0.01512 ↓
RE2	30.0	30.0	192.2	52466	734	12.1	0.78700
	85.0 ↑	90.0 ↑	244.0	10013	738	12.9	0.15619 ↓

(a) Performance-related problems.

Ref.	Acc@1	Acc@3	Time (s)	Tokens In	Tokens Out	Steps	Cost ($)
AG1	0.0	0.0	74.3	6112	1568	30.0	0.04584
	80.0 ↑	80.0 ↑	74.1	14223	1862	16.8	0.11267 ↓
AG2	60.0	60.0	261.0	15355	277	24.6	0.30710
	80.0 ↑	80.0 ↑	311.0	51852	399	20.6	1.04304 ↑
RE1	100.0	100.0	59.0	8790	1318	15.8	0.02198
	80.0 ↓	80.0 ↓	54.6	4172	1092	13.8	0.01643 ↓
RE2	100.0	100.0	172.5	41973	644	12.0	0.62959
	100.0	100.0	210.0	28050	846	15.6	0.42675 ↓

(b) Configuration-related problems

Table 4: Average performance across agents. White rows denote baseline performance with AIOpsLab native tools. Gray rows refer to agents augmented with our custom tools. *Tokens In* are tokens produced by the environment interaction and processed by the LLM, *Tokens Out* are tokens resulting from agent's reasoning.

efficiency, evaluated through the number of reasoning steps, wall-clock time, and output tokens. Higher values in these metrics may correlate with inefficient reasoning, such as redundant or hallucinated trajectories. (3) *Costs*, which depends on the number of input/output tokens consumed by the model.

Tab. 4 shows the results across two performance-related problems and two configuration-related problems. These results highlight that our tools lead to improved accuracy, higher efficiency and reduced costs in most but not all the cases. For example, for performance-related problems `RE2` achieves a 55% increase in `Acc@1` and reduces costs by 5× compared to the baseline. The benefits are mainly due to the fact that `RE2` can access summarized telemetry data, which reduces the number of input tokens consumed by the model (1k vs. 5k), mitigating hallucinations and context-window overflows. In our experience, these were persistent issues with the baseline configuration. We corroborate this explanation with Fig. 1, showing the distribution of tool usage across successful (Fig. 1a) and failed (Fig. 1b) runs, respectively. Success refers to a correct identification of the root-cause service. We observe that *(i)* `get_logs` and `exec_shell` are always privileged by the agents, however *(ii)* in terms of effectiveness our tools have stronger correlation with success, being invoked more frequently in successful RCA compared to failed RCA – i.e., relative increase in usage pattern across Fig. 1a and Fig. 1b. Among the AaaT tools, the impact of `time_series_expert` is less pronounced than `traces_expert`. We attribute this to the use of a vanilla `GPT-4o-mini` model, which we found to be easily overwhelmed by time-series data. We plan to adopt ChatTS [55] in our future work.

Takeaway. Augmenting LLM agents with domain-specific tools—especially those that summarize or analyze telemetry—significantly improves diagnostic accuracy, efficiency, and reduces cost. Effective tool design and integration are critical for enabling LLM agents to focus on relevant signals and avoid context overload.

4.2 The Bad

4.2.1 Overconfidence on shallow initial fixes. In other cases, agents quickly apply shallow remedies yet fail to identify the root cause, resulting in failed RCA. Fig. 3 illustrates an execution involving a misconfiguration in the Kubernetes deployment for the `user-service` microservice, where the replica count was set to zero. The agent first examines the `compose-post-service` and detects connection errors to the `user-service` in the logs. Then, it correctly inspects the Kubernetes deployment and discovers that `user-service` has zero active pod replicas. The agent scales the service back up to one replica, temporarily restoring functionality. However,

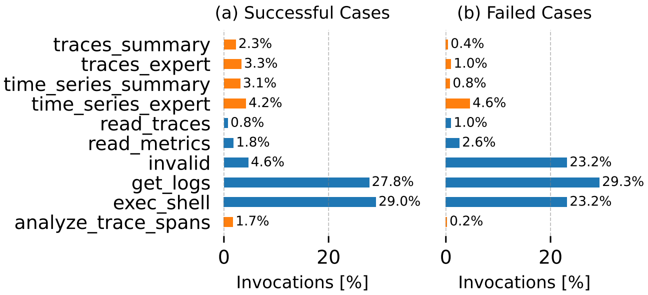

Figure 1: Distribution of tool invocations. Our tools are shown in orange, and used more frequently by the agents during successful executions.

despite this fix, the agent decides not to further investigate the underlying cause (misconfigured Kubernetes manifest) and reports no issues to the user. As a consequence, the user will face the same problem again when the service deployment is restarted.

4.2.2 Erroneous and inconsistent use of APIs. Despite the set of available tools is explicitly listed into the initial system prompt, we observe malformed or hallucinated tool invocations, incorrect parameter handling, and even subsequent repeated occurrences of the same erroneous invocations. Hallucinated tool invocations include `fetch_data`, `k8s_get_pods`, `plot_metric`, and similar. We observe they occur more frequently with the `AG1` and `AG2` architectures. We quantified the impact of these issues in Fig. 1. Erroneous invocations can occur even in successful runs, accounting for ∼5% of the total tool invocations (Fig. 1a), and are quite common in failed runs (Fig. 1b). However, comparing the two, we observe more than a 4× increase in the number of erroneous invocations in failed runs compared to successful runs, providing strong evidence that incorrect and inconsistent API usage significantly degrades diagnostic performance. In practice, we found that agents lack error awareness. In many cases, agents persist with the erroneous action, despite having already observed the action leading to execution errors, without adapting or invoking alternative strategies (e.g., repeated calls to `plot_metric`).

Takeaway. LLM agents are prone to overconfidence, shallow fixes, and repeated tool misuse, especially when lacking error awareness or self-correction mechanisms. Ensuring robust error handling and strategy adaptation is essential to prevent diagnostic failures and improve reliability.

4.3 The Ugly

Finally, we highlight a subset of the most severe issues we observed in our experiments, where agents take actions that are *(i)* unnecessary, *(ii)* ineffective and *(iii)* potentially harmful to the system.

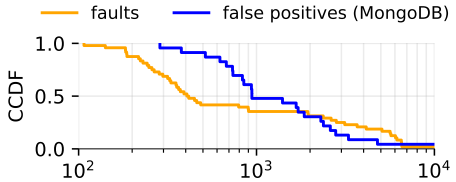

Figure 2: Tokens/steps ratio. Agents may process more context for fault-free executions than for incidents.

4.3.1 The chaotic consequences of false positives. Here we look at executions where no failure is injected. In these cases, we expect the agent to report that the system is healthy with relatively low effort. To quantify this, we compute the ratio between the input tokens and the number of reasoning steps for each agent execution. We expect the ratio to be generally lower for healthy executions than for faulty executions. Interestingly, by filtering cases for which this is not true, we encounter *ugly* behaviors, such as the one shown in Fig. 2. The figure compares the complementary cumulative distribution function (CCDF) of the token-per-steps ratio for two scenarios. The yellow curve refers to all executions where we inject a failure in the system (faulty). The blue curve refers to *healthy* executions where, however, the `user-mongodb` container starts *after* its client container, i.e., `user-timeline-service`, leading to repeated connection errors and retries until the database container is ready. Fig. 4 reports the logs of the `user-timeline-service`, from which a SRE would easily recognize all services started correctly (last line). Fig. 2 shows that this scenario consumes around 5× more tokens-per-step on average compared to faulty executions. We inspected these executions and found that the agent spends a significant amount of steps for connectivity checks with utilities like `ping`, `curl`, `nc`, `telnet`, `nslookup`, or checking Kubernetes network policies, or a combination of these. In the most absurd cases, the agent tries to patch the `coredns` service, which is not related to the problem, or to create an index in the MongoDB database. A sample of these executions is included in Appendix A. Only *once* the agent correctly concluded that the system is healthy and suggested to add a readiness probe to the Kubernetes manifest file (Fig. 7), but even in this case it . In all other cases, Overall, the agents tend to excessively focus on telemetry signals (e.g, `user-mongodb` logs), often getting overwhelmed by noisy symptoms and failing to reason about basic relationships.

4.3.2 Illogical sequence for straightforward tool chains. We observe that agents can waste tokens by issuing actions in illogical sequence, even when the correct invocation order is straightforward. For example, the agents begins its investigation by calling `get_logs("test-hotel-res", "service-name")`, treating the literal string "service-name" as if it were a valid identifier. Surprisingly, we observed this behavior occurs in as many as 28 executions, despite the agents could retrieve available services and have access to a well-defined tool description – e.g., via `kubectl get pods` and tool's docstring documentation (Listing 1), respectively.

Takeaway. Without proper planning, LLM agents can waste resources, take unnecessary or even harmful actions, and fail to recognize obvious solutions. Addressing hallucinations, improving planning, and enforcing guardrails are necessary for safe and effective autonomous RCA.

5 Conclusion

Large Language Models offer a compelling vision for fully automating RCA in complex microservice environments. We examined the extent to which this vision holds up in practice, documenting what works, what breaks, and what needs fixing. Our study shows that LLM agents can assist in useful ways and surfaces a range of limitations, such as redundant and logically inconsistent tool use, or persistent misunderstandings of simple system state, all leading to chaotic system troubleshooting. We view this work as a first step toward making LLM-driven RCA more principled and less brittle.

References

[1] Prometheus Authors 2014-2025. 2023. Prometheus: Monitoring system & time series database. https://prometheus.io/.

[2] Ibrahim Abdelaziz, Kinjal Basu, Mayank Agarwal, Sadhana Kumaravel, Matthew Stallone, Rameswar Panda, Yara Rizk, GP Bhargav, Maxwell Crouse, Chulaka Gunasekara, Shajith Ikbal, Sachin Joshi, Hima Karanam, Vineet Kumar, Asim Munawar, Sumit Neelam, Dinesh Raghu, Udit Sharma, Adriana Meza Soria, Dheeraj Sreedhar, Praveen Venkateswaran, Merve Unuvar, David Cox, Salim Roukos, Luis Lastras, and Pavan Kapanipathi. 2024. Granite-Function Calling Model: Introducing Function Calling Abilities via Multi-task Learning of Granular Tasks. arXiv:2407.00121 [cs.LG] https://arxiv.org/abs/2407.00121

[3] Leila Abdollahi Vayghan, Mohamed Aymen Saied, Maria Toeroe, and Ferhat Khendek. 2018. Deploying Microservice Based Applications with Kubernetes: Experiments and Lessons Learned. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD '18)*.

[4] Toufique Ahmed, Supriyo Ghosh, Chetan Bansal, Thomas Zimmermann, Xuchao Zhang, and Saravan Rajmohan. 2023. Recommending Root-Cause and Mitigation Steps for Cloud Incidents Using Large Language Models. In *International Conference on Software Engineering*. 1737–1749.

[5] Sarah Alnegheimish, Linh Nguyen, Laure Berti-Equille, and Kalyan Veeramachaneni. 2024. Large language models can be zero-shot anomaly detectors for time series? arXiv:2405.14755 [cs.LG] https://arxiv.org/abs/2405.14755

[6] Vaastav Anand, Deepak Garg, Antoine Kaufmann, and Jonathan Mace. 2023. Blueprint: A Toolchain for Highly-Reconfigurable Microservice Applications. In *SOSP*. Association for Computing Machinery.

[7] Vaastav Anand, Pedro Las-Casas, Rodrigo Fonseca, and Antoine Kaufmann. 2025. Towards Using Llms for Distributed Trace Comparison (Abstract) . In *2025 IEEE/ACM International Workshop on Cloud Intelligence & AIOps (AIOps)*. IEEE Computer Society.

[8] Anthropic. 2025. Claude: An AI Assistant for Collaborative Reasoning. urlhttps://www.anthropic.com/claude.

[9] The Jaeger Authors. 2023. Jaeger. https://www.jaegertracing.io/.

[10] Kinjal Basu, Ibrahim Abdelaziz, Kiran Kate, Mayank Agarwal, Maxwell Crouse, Yara Rizk, Kelsey Bradford, Asim Munawar, Sadhana Kumaravel, Saurabh Goyal, Xin Wang, Luis A. Lastras, and Pavan Kapanipathi. 2025. NESTFUL: A Benchmark for Evaluating LLMs on Nested Sequences of API Calls. arXiv:2409.03797 [cs.AI] https://arxiv.org/abs/2409.03797

[11] Yinfang Chen, Huaibing Xie, Minghua Ma, Yu Kang, Xin Gao, Liu Shi, Yunjie Cao, Xuedong Gao, Hao Fan, Ming Wen, et al. 2023. Empowering Practical Root Cause Analysis by Large Language Models for Cloud Incidents. *arXiv preprint arXiv:2305.15778* (2023).

[12] Yinfang Chen, Huaibing Xie, Minghua Ma, Yu Kang, Xin Gao, Liu Shi, Yunjie Cao, Xuedong Gao, Hao Fan, Ming Wen, Jun Zeng, Supriyo Ghosh, Xuchao Zhang, Chaoyun Zhang, Qingwei Lin, Saravan Rajmohan, Dongmei Zhang, and Tianyin Xu. 2024. Automatic Root Cause Analysis via Large Language Models for Cloud Incidents. In *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys'24)*.

[13] Microsoft Corporation. 2025. AIOpsLab: GitHub Repository. https://github.com/microsoft/AIOpsLab.

[14] Manqing Dong, Hao Huang, and Longbing Cao. 2024. Can LLMs Serve As Time Series Anomaly Detectors? arXiv:2408.03475 [cs.LG] https://arxiv.org/abs/2408.03475

[15] Chris Egersdoerfer, Di Zhang, and Dong Dai. 2023. Early Exploration of Using ChatGPT for Log-based Anomaly Detection on Parallel File Systems Logs. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing* (Orlando, FL, USA) *(HPDC '23)*. Association for Computing Machinery, New York, NY, USA, 315–316. doi:10.1145/3588195.3595943

[16] Úlfar Erlingsson, Marcus Peinado, Simon Peter, and Mihai Budiu. 2011. Fay: Extensible Distributed Tracing from Kernels to Clusters. In *SOSP* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 311–326. doi:10.1145/2043556.2043585

[17] Rodrigo Fonseca, George Porter, Randy H. Katz, and Scott Shenker. 2007. X-Trace: A Pervasive Network Tracing Framework. In *USENIX NSDI*. USENIX Association.

[18] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. 2021. Sage: practical and scalable ML-driven performance debugging in microservices *(ASPLOS '21)*. Association for Computing Machinery, 135–151.

[19] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. Association for Computing Machinery.

[20] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. 2019. Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices. In *ASPLOS* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 19–33. doi:10.1145/3297858.3304004

[21] C. W. J. Granger. 1969. Investigating Causal Relations by Econometric Models and Cross-spectral Methods. *Econometrica* 37, 3 (1969), 424–438.

[22] Wei Guan, Jian Cao, Shiyou Qian, Jianqi Gao, and Chun Ouyang. 2025. LogLLM: Log-based Anomaly Detection Using Large Language Models. arXiv:2411.08561 [cs.SE] https://arxiv.org/abs/2411.08561

[23] Saurabh Jha, Rohan Arora, Yuji Watanabe, Takumi Yanagawa, Yinfang Chen, Jackson Clark, Bhavya Bhavya, Mudit Verma, Harshit Kumar, Hirokuni Kitahara, Noah Zheutlin, Saki Takano, Divya Pathak, Felix George, Xinbo Wu, Bekir O. Turkkan, Gerard Vanloo, Michael Nidd, Ting Dai, Oishik Chatterjee, Pranjal Gupta, Suranjana Samanta, Pooja Aggarwal, Rong Lee, Pavankumar Murali, Jae wook Ahn, Debanjana Kar, Ameet Rahane, Carlos Fonseca, Amit Paradkar, Yu Deng, Pratibha Moogi, Prateeti Mohapatra, Naoki Abe, Chandrasekhar Narayanaswami, Tianyin Xu, Lav R. Varshney, Ruchi Mahindru, Anca Sailer, Laura Shwartz, Daby Sow, Nicholas C. M. Fuller, and Ruchir Puri. 2025. ITBench: Evaluating AI Agents across Diverse Real-World IT Automation Tasks. arXiv:2502.05352 [cs.AI] https://arxiv.org/abs/2502.05352

[24] Yuxuan Jiang, Chaoyun Zhang, Shilin He, Zhihao Yang, Minghua Ma, Si Qin, Yu Kang, Yingnong Dang, Saravan Rajmohan, Qingwei Lin, and Dongmei Zhang. 2023. Xpert: Empowering Incident Management with Query Recommendations via Large Language Models. arXiv:2312.11988 [cs.SE] https://arxiv.org/abs/2312.11988

[25] Yuxuan Jiang, Chaoyun Zhang, Shilin He, Zhihao Yang, Minghua Ma, Si Qin, Yu Kang, Yingnong Dang, Saravan Rajmohan, Qingwei Lin, and Dongmei Zhang. 2024. Xpert: Empowering Incident Management with Query Recommendations via Large Language Models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE'24)*.

[26] Pengxiang Jin, Shenglin Zhang, Minghua Ma, Haozhe Li, Yu Kang, Liqun Li, Yudong Liu, Bo Qiao, Chaoyun Zhang, Pu Zhao, Shilin He, Federica Sarro, Yingnong Dang, Saravan Rajmohan, Qingwei Lin, and Dongmei Zhang. 2023. Assess and Summarize: Improve Outage Understanding with Large Language Models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1657–1668. doi:10.1145/3611643.3613891

[27] Alex Karpenko, Alexander Wei, Allison Tam, Ananya Kumar, Andre Saraiva, Andrew Kondrich, Andrey Mishchenko, Ashvin Nair, Behrooz Ghorbani, Bohan Zhang, Brandon McKinzie, Brydon Eastman, Chak Ming Li, Chris Koch, Dan Roberts, David Dohan, David Mely, Dimitris Tsipras, Enoch Cheung, Eric Wallace, Hadi Salman, Haiming Bao, Hessam Bagherinezhad, Ilya Kostrikov, Jiacheng Feng, John Rizzo, Karina Nguyen, Kevin Lu, Kevin Stone, Lorenz Kuhn, Mason Meyer, Mikhail Pavlov, Nat McAleese, Oleg Boiko, Oleg Murk, Peter Zhokhov, Randall Lin, Raz Gaon, Rhythm Garg, Roshan James, Rui Shu, Scott McKinney, Shibani Santurkar, Suchir Balaji, Taylor Gordon, Thomas Dimson, and Weiyi Zheng. 2025. Learning to reason with LLMs. https://openai.com/index/learning-to-reason-with-llms/.

[28] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. 2024. DSPy: Compiling Declarative Language Model Calls into State-of-the-Art Pipelines. In *International Conference on Learning Representations (ICLR '24)*. OpenReview.net.

[29] Myunghwan Kim, Roshan Sumbaly, and Sam Shah. 2013. Root Cause Detection in a Service-Oriented Architecture. *SIGMETRICS Perform. Eval. Rev.* (2013).

[30] Grafana Labs. 2023. Grafana Labs Observability Survey 2023. https://grafana.com/about/press/2023/03/08/grafana-labs-observability-survey-2023-finds-centralization-saves-time-and-money-for-an-industry-plagued-by-tool-and-data-source-overload/.

[31] Pedro Las-Casas, Alok Gautum Kumbhare, Rodrigo Fonseca, and Sharad Agarwal. 2024. LLexus: an AI agent system for incident management. *SIGOPS Oper. Syst. Rev.* 58, 1 (Aug. 2024).

[32] Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. 2023. CAMEL: Communicative Agents for "Mind" Exploration of Large Language Model Society. In *NeurIPS 2023*. Curran Associates Inc.

[33] Zeyan Li, Junjie Chen, Rui Jiao, Nengwen Zhao, Zhijun Wang, Shuwei Zhang, Yanjun Wu, Long Jiang, Leiqin Yan, Zikai Wang, Zhekang Chen, Wenchi Zhang, Xiaohui Nie, Kaixin Sui, and Dan Pei. 2021. Practical Root Cause Localization for Microservice Systems via Trace Analysis. In *2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS)*. 1–10. doi:10.1109/IWQOS52092.2021.9521340

[34] Jun Liu, Chaoyun Zhang, Jiaxu Qian, Minghua Ma, Si Qin, Chetan Bansal, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. 2024. Large Language Models can Deliver Accurate and Interpretable Time Series Anomaly Detection. arXiv:2405.15370 [cs.CL] https://arxiv.org/abs/2405.15370

[35] Yuhe Liu, Changhua Pei, Longlong Xu, Bohan Chen, Mingze Sun, Zhirui Zhang, Yongqian Sun, Shenglin Zhang, Kun Wang, Haiming Zhang, Jianhui Li, Gaogang Xie, Xidao Wen, Xiaohui Nie, Minghua Ma, and Dan Pei. 2024. OpsEval: A Comprehensive IT Operations Benchmark Suite for Large Language Models. arXiv:2310.07637 [cs.AI] https://arxiv.org/abs/2310.07637

[36] Meta. 2025. The Llama 4 herd: The beginning of a new era of natively multimodal AI innovation. urlhttps://ai.meta.com/blog/llama-4-multimodal-intelligence/.

[37] OpenAI. 2023. GPT-4 Technical Report. urlhttps://cdn.openai.com/papers/gpt-4.pdf.

[38] OpenAI. 2025. OpenAI Models. https://platform.openai.com/docs/models.

[39] John Paparrizos and Luis Gravano. 2016. k-Shape: Efficient and Accurate Clustering of Time Series. *SIGMOD Rec.* 45, 1 (June 2016).

[40] Austin Parker. 2025. MCP Easy as 1-2-3? urlhttps://www.honeycomb.io/blog/mcp-easy-as-1-2-3?utm_source=chatgpt.com.

[41] Luan Pham, Hongyu Zhang, Huong Ha, Flora Salim, and Xiuzhen Zhang. 2025. RCAEval: A Benchmark for Root Cause Analysis of Microservice Systems with Telemetry Data. In *The 2025 ACM Web Conference (WWW)*.

[42] Devjeet Roy, Xuchao Zhang, Rashi Bhave, Chetan Bansal, Pedro Las-Casas, Rodrigo Fonseca, and Saravan Rajmohan. 2024. Exploring LLM-Based Agents for Root Cause Analysis. In *ACM FSE*. Association for Computing Machinery.

[43] Keshav Santhanam, Omar Khattab, Jon Saad-Falcon, Christopher Potts, and Matei Zaharia. 2022. ColBERTv2: Effective and Efficient Retrieval via Lightweight Late Interaction. arXiv:2112.01488 [cs.IR] https://arxiv.org/abs/2112.01488

[44] Vishwanath Seshagiri, Siddharth Balyan, Vaastav Anand, Kaustubh Dhole, Ishan Sharma, Avani Wildani, José Cambronero, and Andreas Züfle. 2024. Chatting with Logs: An exploratory study on Finetuning LLMs for LogQL. arXiv:2412.03612 [cs.DB] https://arxiv.org/abs/2412.03612

[45] Manish Shetty, Yinfang Chen, Gagan Somashekar, Minghua Ma, Yogesh Simmhan, Xuchao Zhang, Jonathan Mace, Dax Vandevoorde, Pedro Las-Casas, Shachee Mishra Gupta, Suman Nath, Chetan Bansal, and Saravan Rajmohan. 2024. Building AI Agents for Autonomous Clouds: Challenges and Design Principles. In *ACM Symposium on*

Cloud Computing (SoCC ’24). Association for Computing Machinery.

[46] Yongqian Sun, Jiaju Wang, Zhengdan Li, Xiaohui Nie, Minghua Ma, Shenglin Zhang, Yuhe Ji, Lu Zhang, Wen Long, Hengmao Chen, Yongnan Luo, and Dan Pei. 2025. AIOpsArena: Scenario-Oriented Evaluation and Leaderboard for AIOps Algorithms in Microservices. In *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 809–813. doi:10.1109/SANER64311.2025.00082

[47] Jörg Thalheim, Antonio Rodrigues, Istemi Ekin Akkus, Pramod Bhatotia, Ruichuan Chen, Bimal Viswanath, Lei Jiao, and Christof Fetzer. 2017. Sieve: Actionable insights from monitored metrics in distributed systems. In *ACM/IFIP/USENIX Middleware Conference*. 14–27.

[48] Santiago Valdarrama. 2024. X Post. https://x.com/svpino/status/1841461406081626296?s=46&t=1-9-9RQXPlSV9HwI8l0BGg.

[49] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Ji Heng. 2024. CodeAct: Your LLM Agent Acts Better when Generating Code. In *ICML*. https://arxiv.org/abs/2402.01030

[50] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024. Executable code actions elicit better LLM agents. In *ICML*. JMLR.org, Article 2054.

[51] Zefan Wang, Zichuan Liu, Yingying Zhang, Aoxiao Zhong, Jihong Wang, Fengbin Yin, Lunting Fan, Lingfei Wu, and Qingsong Wen. 2024. RCAgent: Cloud Root Cause Analysis by Autonomous Agents with Tool-Augmented Large Language Models. In *ACM Conference on Information and Knowledge Management (CIKM ’24)*. ACM.

[52] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *NeurIPS*. Curran Associates Inc.

[53] Li Wu, Johan Tordsson, Erik Elmroth, and Odej Kao. 2020. MicroRCA: Root cause localization of performance issues in microservices. In *IEEE NOMS*.

[54] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. 2023. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation. arXiv:2308.08155 [cs.AI] https://arxiv.org/abs/2308.08155

[55] Zhe Xie, Zeyan Li, Xiao He, Longlong Xu, Xidao Wen, Tieying Zhang, Jianjun Chen, Rui Shi, and Dan Pei. 2025. ChatTS: Aligning Time Series with LLMs via Synthetic Data for Enhanced Understanding and Reasoning. arXiv:2412.03104 [cs.AI] https://arxiv.org/abs/2412.03104

[56] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. arXiv:2210.03629

[57] Guangba Yu, Pengfei Chen, Hongyang Chen, Zijie Guan, Zicheng Huang, Linxiao Jing, Tianjun Weng, Xinmeng Sun, and Xiaoyun Li. 2021. Microrank: End-to-end latency issue localization with extended spectrum analysis in microservice environments. In *ACM Web Conference 2021*. 3087–3098.

[58] Zhaoyang Yu, Minghua Ma, Chaoyun Zhang, Si Qin, Yu Kang, Chetan Bansal, Saravan Rajmohan, Yingnong Dang, Changhua Pei, Dan Pei, Qingwei Lin, and Dongmei Zhang. 2024. MonitorAssistant: Simplifying Cloud Service Monitoring via Large Language Models. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE’24)*.

[59] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. 2010. SherLog: Error Diagnosis by Connecting Clues from Run-Time Logs. In *ACM ASPLOS*. Association for Computing Machinery.

[60] Chenxi Zhang, Xin Peng, Chaofeng Sha, Ke Zhang, Zhenqing Fu, Xiya Wu, Qingwei Lin, and Dongmei Zhang. 2022. DeepTraLog: Trace-Log Combined Microservice Anomaly Detection through Graph-based Deep Learning. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 623–634. doi:10.1145/3510003.3510180

[61] Chenxi Zhang, Bicheng Zhang, Dingyu Yang, Xin Peng, Miao Chen, Senyu Xie, Gang Chen, Wei Bi, and Wei Li. 2025. PromAssistant: Leveraging Large Language Models for Text-to-PromQL. arXiv:2503.03114 [cs.SE] https://arxiv.org/abs/2503.03114

[62] Lei Zhang, Zhiqiang Xie, Vaastav Anand, Ymir Vigfusson, and Jonathan Mace. 2023. The Benefit of Hindsight: Tracing Edge-Cases in Distributed Systems. In *USENIX NSDI*. USENIX Association.

[63] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Dewei Liu, Qilin Xiang, and Chuan He. 2019. Latent error prediction and fault localization for microservice applications by learning from system trace logs. In *ACM ESEC/FSE*. Association for Computing Machinery.

A Execution examples

The following figures illustrate a range of LLM agent behaviors observed during root cause analysis tasks on the AIOpsLab platform. These include both successful diagnoses and common failure patterns. In some cases, agents correctly identify actionable issues – such as a missing readiness probe (Fig. 7). Other examples highlight more problematic behaviors: repeatedly misinterpreting stale logs as ongoing failures (Fig. 4), issuing unnecessary or ineffective actions like creating MongoDB indexes without justification (Fig. 5), or attempting to patch CoreDNS based on incorrect assumptions about DNS failures (Fig. 6). These traces expose a mix of capabilities and limitations, underscoring the need for stronger temporal reasoning, action planning, and state grounding in LLM-based diagnostic agents.

```
Agent: Starting investigation at compose-post-service.

Actions: get_logs("test-social-network", "compose-post-service")

Agent: Logs show connection refusals to user-service.

Actions:
• kubectl get pods -n test-social-network
• kubectl get all -n test-social-network | grep user
• kubectl describe deployment user-service -n test-social-network

Agent: Kubernetes inspection reveals user-service is deployed but scaled to zero replicas. I will scale it back to one replica.

Actions: kubectl scale deployment user-service -replicas=1 -n test-social-network

Framework: Pod user-service is active.

Agent: Pod for user-service is now running. Seems like the issue is fixed.

Actions: kubectl get pods -n test-social-network

Agent: All pods are running with no issues. Submitting: [None].
```

Figure 3: In this execution, the agent correctly identifies a deployment issue in the Kubernetes deployment: the user-service microservice was scaled to zero replicas. However, the agent misses the root-cause: a misconfiguration in the Kubernetes manifest file to deploy user-service. Thus, it failed the RCA localization task.

```
[14:08:26.077189] <error>: (utils_mongodb.h:76:CreateIndex) Error in createIndexes: No suitable servers found: ‘serverSelectionTimeoutMS‘ expired: [unknown connection error calling ismaster on ’user-timeline-mongodb:27017’]
[14:08:26.079309] <error>: (UserTimelineService.cpp:103:main) Failed to create mongodb index, try again
[14:08:27.378146] <error>: (utils_mongodb.h:76:CreateIndex) Error in createIndexes: No suitable servers found: ‘serverSelectionTimeoutMS‘ expired: [unknown connection error calling ismaster on ’user-timeline-mongodb:27017’]
[14:08:27.378263] <error>: (UserTimelineService.cpp:103:main) Failed to create mongodb index, try again
[14:08:28.677149] <error>: (utils_mongodb.h:76:CreateIndex) Error in createIndexes: No suitable servers found: ‘serverSelectionTimeoutMS‘ expired: [unknown connection error calling ismaster on ’user-timeline-mongodb:27017’]

                    Repeating until user-timeline-mongodb kubernetes pod is up

[14:08:47.943816] <info>: (UserTimelineService.cpp:148:main) Starting the user-timeline-service server...
```

Figure 4: Error logs in user-timeline-service, caused by a late start of the MongoDB database pod. The errors are non-critical, and could be prevented by proper handling of startup dependencies in the Kubernetes deployment.

```
Environment: ... user-timeline-service pod experiencing restarts ...
Actions: get_logs("test-social-network", "user-timeline-service")
Agent: Logs (reported in Fig. 4) show repeated MongoDB connection failures. Checking MongoDB pod status.
Actions: kubectl get pods -n test-social-network -l app=user-timeline-mongodb
Agent: The ‘user-timeline-mongodb‘ pod is running and has not restarted, which suggests that the pod itself is not crashing. However, the connection issues reported in the logs of the ‘user-timeline-service‘ could be due to network issues or misconfigurations in the service or MongoDB setup. I should check the network policies or service configurations to ensure that the ‘user-timeline-service‘ can properly connect to the ‘user-timeline-mongodb‘
Actions: kubectl describe service user-timeline-mongodb -n test-social-network
Agent: The user-timeline-mongodb service is configured correctly with a ClusterIP and the correct port. The endpoint is also correctly set. Since there are no events indicating issues, the problem might be related to network policies or firewall rules that are preventing the ‘user-timeline-service‘ from connecting to the MongoDB service. I should check if there are any network policies in place that might be blocking the connection.
Actions: kubectl get networkpolicies -n test-social-network"
Environment: empty policies
Agent: The output for the network policies command is missing, which suggests that there might be no network policies defined in the ‘test-social-network‘ namespace. I should check the MongoDB logs for any errors or warnings that might provide more insight into the connection issues.
Actions: mongo –eval "db.user_timeline.getIndexes()"
Agent: The MongoDB logs indicate that there are no critical errors in the MongoDB startup process, but there is a slow query log entry, which suggests that the ‘user-timeline-service‘ is experiencing performance issues when querying the database. This could be contributing to the high CPU usage observed earlier. The slow query might be due to inefficient indexing or a large dataset being processed. To address this, I should investigate the indexing strategy for the ‘user-timeline‘ collection and consider optimizing the query or adding appropriate indexes to improve performance.
Actions: kubectl exec -n test-social-network user-timeline-mongodb-6758898d48-s78vn – bash -c ’mongo –eval "db.user_timeline.createIndex(user_id: 1)"
Environment: Index creation succeeds.
```

Figure 5: The agent tries to create a new index on the MongoDB service for no reason.

```
Environment: ... user-timeline-service pod experiencing restarts ...
Actions: get_logs("test-social-network", "user-timeline-service")
Agent: Logs (reported in Fig. 4) show repeated MongoDB connection failures. Suspecting DNS issues.
Actions: kubectl logs -n kube-system -l k8s-app=kube-dns
Agent: CoreDNS logs show multiple DNS timeouts. Attempting to patch CoreDNS to fix resolution.
Actions: kubectl patch configmap coredns -n kube-system –type merge -p ...
Agent: Patch failed repeatedly with API errors and malformed payloads. Trying to create coredns-patch.yaml as fallback.
Actions: kubectl apply -f coredns-patch.yaml && kubectl rollout restart deployment coredns -n kube-system
Agent: YAML file not found, patch attempt unsuccessful. DNS issue remains unresolved.
Actions: continues connectivity checks ···
                         Agent ran out of steps
```

Figure 6: In this execution, the agent attempts to mitigate MongoDB connection errors by patching CoreDNS configuration. The action is unnecessary and repeatedly fails due to malformed patch commands and incorrect assumptions about the DNS infrastructure.

user-service is unable to connect to user-mongodb (Fig. 4).

Agent. Since there was no output provided for the network policies, it may indicate there are no network policies defined or there was an error in retrieving them. I will check the connectivity between the user-service and user-mongodb

Actions. kubectl exec -n test-social-network $(kubectl get pods -n test-social-network -l app=user-service -o jsonpath='.items[0].metadata.name') -- ping -c 4 user-mongodb

Environment: command terminated with exit code 127

Agent. The exit code 127 indicates that the ‘ping‘ command is not available in the user-service pod’s environment. I will use a different approach

Actions. kubectl exec -n test-social-network $(kubectl get pods -n test-social-network -l app=user-service -o jsonpath='.items[0].metadata.name') -- nc -zv user-mongodb 27017

Environment: command terminated with exit code 127

Agent. The exit code 127 indicates that the ‘nc‘ command is not available in the user-service pod’s environment. I will use a different approach

Actions. kubectl run debug –rm -i –tty –image=busybox –restart=Never –namespace=test-social-network – wget -qO- user-mongodb:27017

Environment:It looks like you are trying to access MongoDB over HTTP on the native driver port.

Agent. The response indicates that the connection to user-mongodb was successful, but the attempt to access it via HTTP on the MongoDB native driver port was incorrect. This confirms that the network connectivity is not the issue. Since user-service is unable to connect to user-mongodb, it might be a configuration issue within the user-service itself. I will check the configuration of the user-service to ensure it is correctly set up to connect to the MongoDB service. The issue might be related to the configuration files mounted from the ConfigMap. I inspect the contents of the service-config.json to verify the MongoDB connection settings.

Actions. kubectl get pods -n test-social-network -l app=user-service -o jsonpath='.items[0].metadata.name') -- cat /social-network-microservices/config/service-config.json

Environment: ...

```
"user-mongodb": {
  "addr": "user-mongodb",
  "port": 27017,
  "connections": 512,
  "timeout_ms": 10000,
  "keepalive_ms": 10000
},
\( \ldots \)
```

Agent. Config uses user-mongodb:27017, The configuration file specifies the MongoDB service address as üser-mongodbänd port as 27017, I will check if environment variables align.

Agent continues basic checks (e.g., DNS resolution, environment, config files) for several steps ...

Actions. get_logs("test-social-network", "user-service")

Agent. Logs still show CreateIndex failed: connection refused. Checking MongoDB readiness configuration.

Actions. kubectl get deployment user-mongodb -o yaml

Agent. The deployment configuration for user-mongodb does not include any readiness or liveness probes, which could lead to the service being marked as available before it is fully ready to handle requests. Adding a readiness probe could help ensure that the user-service only attempts to connect to MongoDB once it is ready.

Figure 7: In this execution (with agent RE1), the agent exhaustively checks environment variables, pod states, service configs, DNS, and deployment specs—all of which are correct. Finally, it correctly concludes a readiness probe is missing.

B Implementation

B.1 Tools

```
def get_logs(namespace: str, service: str) -> str:
    """
    Collects relevant log data from a pod using Kubectl. Use the service name without the pod suffix.

    Args:
        namespace (str): The namespace in which the service is running.
        service (str): The name of the service.

    Returns:
        str | dict | list[dicts]: Log data as a structured object or a string.
    """
```

Listing 1: Docstrings for the `get_logs` tool.

B.2 MSACausalBench Question Examples

Each question includes its reference answer and is categorized into one of five categories: (1) *System Deployment* - infrastructure components and configurations such as databases, caches, and retry policies; (2) *Service Dependencies* - direct service-to-service invocation relationships at the code level; (3) *Operational Dependencies* - runtime failure analysis examining how service failures impact other services' operations; (4) *Latency Propagation* - causal analysis of performance degradation across services; and (5) *Request Flow* - end-to-end request tracing through the architecture.

Category	Question	Answer
System Deployment	*If GeoService uses a database, which one does it use?*	MongoDB
Service Dependencies	*Which services does FrontEndService call?*	SearchService, ProfileService, RecommendationService, UserService
Operational Dependencies	*Will SearchHandler Start be successful if there is a failure in RateService?*	No
Latency Propagation	*If there is a severe hotspot in GeoService (i.e., increasing latency of its operations), which other services will likely experience increased latency as a result?*	FrontEndService and SearchService

Table 5: Sample Questions from MSACausalBench

Request Flow Analysis.
Question: Please create a Mermaid sequence diagram showing the flow of method calls from the SearchHandler function, assuming all services are functioning normally starting from the client.
Answer:

```
sequenceDiagram
    participant Client
    participant FrontEndService
    participant SearchService
    participant GeoService
    participant RateService
    participant ReservationService
    participant ProfileService
```

```
Client->>FrontEndService:
FrontEndService->>SearchService:
SearchService->>GeoService:
GeoService-->>SearchService:
SearchService->>RateService:
RateService-->>SearchService:
SearchService-->>FrontEndService:
FrontEndService->>ReservationService:
ReservationService-->>FrontEndService:
FrontEndService->>ProfileService:
ProfileService-->>FrontEndService:
FrontEndService-->>Client:
```