ARCA: Agentic Root Cause Analyzer

Ibraheem Mohammed Alsaghier
KAUST

Omar Ayman Fayoumi
KAUST

Abstract

Root cause analysis (RCA) of service-level objective (SLO) violations in microservice architectures presents significant challenges due to the complexity of distributed systems and the cognitive load placed on on-call engineers. This paper introduces ARCA (Agentic Root Cause Analyzer), a novel system that leverages large language models (LLMs) and the ReAct framework to automate and streamline the RCA process. We present CodeReAct, an innovative agent architecture that combines dynamic code execution capabilities with flexible state management to emulate the adaptive investigation patterns of experienced site reliability engineers. To evaluate our approach, we introduce two comprehensive benchmarks: `MSACausalBench` for assessing causal reasoning capabilities in microservice environments, and `SLOvRCABench` for testing end-to-end root cause analysis performance. Our results demonstrate that CodeReAct achieves strong performance in system understanding tasks, with up to 93% accuracy in code-based analysis, while showing promising results in identifying root causes across various failure scenarios. We also provide an open workload orchestration and fault injection library built on Blueprint and Docker to facilitate further research in this domain. Our work highlights both the potential and current limitations of LLM-based agents in automated root cause analysis, while establishing new benchmarks for evaluating such systems. Our work is open sourced and available at https://github.com/alessandrocornacchia/llm-micro

1 Introduction

The microservice architecture has become the *de facto* approach for engineering large-scale applications over the past decade and has seen unanimous adoption by tech companies, including Netflix, Google, and Amazon [18]. In a microservice architecture application, a set of decoupled functional services are developed and tested independently by their respective teams to isolate the complexity of the overall system. They are then integrated with one and another via networking primitives, such as socket connections or Remote Procedure Calls (RPCs). Because of the maximal decoupling, microservice applications can be developed and deployed on different technological stacks and even maintained independently, making them particularly attractive for larger companies [12].

As a result of the many moving parts of microservice applications, an additional maintenance burden is incurred in the case of emergent performance anomalies. These anomalies present as violations of SLOs that are not localizable to any particular service but emerge as a result of a problematic metastable interaction between two or more services. Such anomalies are severe in their effect on end-service and arduous in resolution as they require careful hierarchical and boundary analysis *in between* services [2].

Root Cause Analysis (RCA) is integral to mitigating performance anomalies in microservice applications, yet it is fraught with significant labor costs, potential errors, and immense stress for on-call engineers (OCEs). Traditional RCA methods often demand extensive manual collection and analysis of diverse data sources, including logs, metrics, traces, and incident tickets. This process is not only time-consuming and error-prone but also exacerbated by the "information spectrum" challenge, where engineers must navigate between insufficient data and information overload. These scenarios intensify the pressure on OCEs, who must resolve incidents under critical time constraints to minimize system downtime and operational disruptions. Additionally, troubleshooting guides (TSGs), often employed to streamline RCA, are themselves problematic—overwhelming in volume, frequently outdated, and ill-suited for novel incident types. As a result, OCEs face the dual burden of sifting through complex, voluminous data and maintaining composure under high-pressure conditions, detracting from the primary goal of swiftly resolving incidents and restoring system functionality [21].

Microservice application incidents can prove to be significantly costly to deal with if not addressed correctly. An failure experienced by Meta illustrates this point very well. A 14 hour outage led to a loss of revenue of $90 million. AWS had a similar outage lasting for four hours, which led to an 150 million dollar loss for corporations in the S&P 500 index [26].

Recent work has investigated the deployment of machine learning tools to aid in systems engineering and research processes, including Seer and Sage for microservice application debugging, with the latter being deployed in production systems at Amazon [6, 7]. While these systems are traditionally quantitative, LLMs hold a particular promise for assisting in similar processes that require reasoning over domain-specific knowledge and context [11]. Moreover, as a result of the Chinchilla scaling laws for compute and training data, larger and more capable models are being trained in the industry to streamline ever more sophisticated tasks [9].

We take advantage of this in our system ARCA (Agentic Root Cause Analyzer). We utilize an LLM agent with the ReAct framework

Our main contributions are as follows:

- We provide an open workload orchestration, fault injection, and data collection library in Python which we use to generate data for our evals on top of Blueprint [1] and Docker.
- We propose two sets of open benchmarks for evaluating agentic systems in the SLO violation root cause analysis (RCA) task: `MSACausalBench` and `SLOvRCABench`.
- We develop a novel dynamic-action agent we refer to as CodeReAct that combines the ReAct framework that uses a tool sandbox to diagnose faults in end-to-end failures.

2 Overview

The ARCA project consists of an end-to-end environment for performing automated root cause analysis and contains toolkits enabling researchers to perform the following tasks in a streamlined fashion. We chose to support specific tasks that present the greatest challenges in our perspective for researchers to advance the field of automated RCA in the context of microservice applications. We briefly discuss the guiding principles that led us to choosing the specific tasks and catering the specific tools around them:

1. *Usability:* automated RCA is systems field, and it often serves as a barrier to entry for researchers which are more ML-oriented to interact with the many moving parts of microservice applications and orchestrate experiments in a reproducible fashion without much hinderence. Indeed, this was a great takeaway in our early scaffolding experiences with ARCA.
2. *Modularity:* usability ought not to come at the cost of flexibility. In the context of interacting with microservice applications, researchers ought to be able to work with virtually arbitrary scenarios as long as there is a general guideline to implement such scenarios. Further, extending the orchestrator should port with code researchers already possess. This is especially necessary in the context of reproducing dubious metastable failures.

These platform-level principles guide the overall architecture of ARCA to ensure both accessibility and flexibility. Building upon this foundation, we developed our flagship agent design–CodeReAct–which implements these principles while introducing its own set of important design decisions specifically tailored to the cognitive processes of automated root cause analysis:

1. *Adaptive Investigation Pattern:* in our early experiment with LangChain [20], a major prohibitive challenge was that the agent's flow needed to be strictly dictated–this is by design as it provides greater reliability in production-environment tasks. However, SREs performing RCA on real incidents do not follow an explicit flowchart. Rather, they undulate around a set of artifacts and may revisit things that were already seen if downstream artifacts prove it important. We posit that agents may gain a performance advantage when they are designed with such iterative, unrestricted flows, as they will be able to approach and arrive at a root cause through multiple different pathways.
2. *Dynamic Context Management:* subsequent experiments with LangGraph were able to capture iteration via the implementation of a routing agent, but state management proved to be a problem. Many of the current LLM-agent paradigms assume that the problem being solved has been sufficiently characterized so as to assume a specific definition of a state variable which is appended to in runtime. In RCA, such assumptions come at the cost of iterative processing (running .head on a df to know its shape and then summarizing, for example). Indeed, RCA has not been thoroughly characterized and many Trouble Shooting Guides are employed which use entirely different procedures with unique state to perform RCA. Thus we decide to delegate state management entirely to the agent.
3. *Tool-Native Specialization:* Our agent is designed to leverage existing RCA tools' strengths while focusing on what LLMs do best: pattern recognition, contextual analysis, and hypothesis generation. Rather than implementing new analytical capabilities, it serves as an intelligent orchestrator that knows when and how to employ specialized tools for specific tasks. This approach allows the agent to combine the robust, tested capabilities of existing RCA tools with LLM-powered reasoning to guide the investigation process, interpret results, and synthesize findings across multiple tools and data sources. For

example, instead of implementing time series analysis directly, it intelligently delegates to established monitoring and analysis tools while focusing on the higher-level task of interpreting patterns and correlations across multiple data sources.

Based on these principles, we accomodate two principal tasks within ARCA. First, we provide an instrumentation engine that interfaces with base benchmark microservice applications, including the DeathStarBench applications [8] and SockShop, to fully automate emulated workloads, anomaly injection, and relevant data collection. We build this orchestrator to operate on top of Blueprint and Docker [1, 5]. Workloads are run through HTTP requests with multiple threads with configurable durations and throughputs and anomalies are accommodated through a variety of different "chaos engineering" tooling, including StressNG, Sysbench, TrafficControl, as well as some simple CPP programs for injecting memory and CPU stressors at the Docker container level. Finally, after workloads and anomalies have been injected according to the user configuration, our orchestrator interfaces with the Jaeger, Prometheus, and Docker APIs to all relevant data from the period of the experiment, including application-level traces, system performance metrics, and logs, respectively. It also saves ground truth information for when the anomaly is triggered and with which parameters as well as the user workload outputs for more thorough analyses and optimizations. Using this orchestrator, we generate multiple failure scenarios targeting different hardware of Docker containers to enable a broad exploration of the RCA space for cascading failures.

3 Core Idea

3.1 Driving Challenges

The fundamental challenge in automating root cause analysis for microservice applications lies in replicating the cognitive process of experienced SREs. When investigating incidents, SREs rarely follow a strictly linear investigation pattern. Instead, they:

- Fluidly move between different data sources based on emerging insights
- Revisit previously examined data with new context and hypotheses
- Dynamically adjust their investigation strategy based on observations
- Build and validate complex causal hypotheses across system components

This cognitive flexibility presents three key challenges when automating RCA with LLMs:

- **Context Window Limitations**: The vast amount of observability data from microservices creates significant challenges:
  - Direct ingestion of raw metrics, traces, and logs is impractical due to context window constraints
  - Naively truncating data risks losing critical information
  - The computational cost of processing large contexts reduces model performance [14]
- **Tool Integration Complexity**: LLMs face unique challenges in effectively utilizing analysis tools:
  - Traditional frameworks like LangChain enforce rigid execution flows that limit investigation flexibility
  - Maintaining coherent reasoning while switching between different analysis tools
  - Balancing between tool utilization and LLM reasoning capacity
- **Reasoning Reliability**: LLMs exhibit specific limitations in maintaining reliable investigation paths:
  - Tendency to hallucinate or make unfounded conclusions when processing technical data [23]
  - Difficulty in maintaining consistent investigation context across multiple reasoning steps
  - Challenge in recognizing when to revisit previous analyses with new context

3.2 System Design

3.2.1 ReAct as a Thought-Action-Observation Loop

A core insight of our system builds upon the ReAct paradigm, which frames agent decision processes as iterative Thought-Action-Observation loops. In this paradigm, agents reason about their current understanding of a problem (Thought), take actions to gather more information or test hypotheses (Action), and interpret the results of these actions to update their understanding (Observation). ReAct has proven effective in complex reasoning tasks by enabling agents to maintain coherent investigation paths while adapting to new information [23].

We extend this paradigm to address the challenges of root cause analysis through CodeReAct, which augments the traditional ReAct loop with dynamic, stateful, code execution capabilities. This extension transforms the Action phase from simple tool invocation to flexible code composition, enabling the agent to craft precise analytical steps as needed without being too constricted to follow predetermined steps. The augmented loop better mirrors how SREs approach incident investigation–they don't simply follow predefined playbooks but actively compose and refine their analytical approach based on emerging insights.

3.2.2 Adaptive Investigation Pattern

CodeReAct's primary innovation comes from its departure from rigid investigation flows. Traditional automated approaches, including standard ReAct implementations, must constrain agents to predefined tool sets and execution patterns and state management. Instead, CodeReAct enables genuine analytical flexibility by treating code execution as its primary action mechanism. This allows the agent to dynamically compose analysis steps, create custom data transformations, and build intermediate analytical artifacts that inform further investigation.

The ability to execute arbitrary code fundamentally changes how the agent interacts with observability data. Rather than being limited by predefined APIs or tool interfaces, the agent can precisely scope its analysis, combine data sources in novel ways, and create targeted visualizations that illuminate specific aspects of the system's behavior. This flexibility proves especially valuable when investigating complex microservice failures, where the path to root cause often requires creative combination of different analytical approaches.

3.2.3 Dynamic State Management

A key design decision in CodeReAct is the delegation of state management to the agent itself. Rather than maintaining rigid state definitions through PyDantic templates or predetermined investigation paths, the agent's context evolves organically through the investigation process. This approach enables the agent to maintain and explore multiple working hypotheses simultaneously, integrate insights from different analytical paths, and revisit previous findings with new context–much like how experienced SREs operate.

3.2.4 Tool-Native Intelligence

Instead of reimplementing analytical capabilities, CodeReAct acts as an reasoning orchestrator of existing RCA tools. The agent's power comes not from new analytical methods but from its ability to creatively combine established tools through code execution. This approach leverages the robust capabilities of existing analysis tools while adding the layer of strategic thinking necessary for effective root cause analysis.

This design philosophy once again mirrors how experienced SREs approach incident investigation–they don't simply execute tools in sequence but strategically combine their capabilities to validate hypotheses and uncover root causes. We provide CodeReAct with a base set of tools operating on all the artifacts we generate through the orchestrator with read and analyze primitives. We also provide it with access to base Python module APIs that SREs may employ in their investigation as well as more advanced statistical tooling.

Through this synthesis of dynamic code execution, flexible context management, and intelligent tool orchestration, CodeReAct provides a framework for automated root cause analysis that captures the cognitive flexibility of human investigators while maintaining the scalability and reproducibility benefits of automation.

4 Implementation

CodeReAct is implemented as a DSPy module that orchestrates LLM-driven root cause analysis through dynamic code execution. We detail the key components and mechanisms that enable its adaptive investigation capabilities.

4.1 Core Architecture

The core of CodeReact consists of three main components: the thought-action-observation loop manager, the code execution environment, and the trajectory tracking system. The loop manager handles the iterative investigation process, with each iteration comprising:

1. A thought phase where the LLM reasons about the current state and determines the next analytical step
2. An action phase where Python code is dynamically composed and executed
3. An observation phase where results are interpreted and integrated into the investigation context

This loop continues until either a conclusive root cause is identified or the maximum iteration limit is reached. The system employs DSPy's ChainOfThought module for both the reasoning steps and final report generation, enabling explicit tracking of the agent's decision-making process.

4.2 Dynamic Code Execution

To enable flexible analysis, CodeReact executes code through a Jupyter kernel environment that maintains session state across iterations. This approach offers several advantages:

1. It allows for stateful analysis where intermediate results and data structures persist throughout the investigation. The agent can reference and build upon previous computations, enabling incremental refinement of its analysis.
2. The Jupyter environment provides rich output capture capabilities, allowing the agent to generate and interpret statistical summaries, and complex data transformations. This is particularly valuable for understanding temporal patterns in metrics and identifying correlations across different data sources.

3. The environment supports dynamic import of additional libraries and definition of helper functions, giving the agent the ability to extend its analytical capabilities as needed during the investigation.

4.3 Error Recovery and Iteration Management

Root cause analysis often involves exploratory data analysis where not all approaches yield useful results. CodeReact implements a robust error handling system with multiple recovery mechanisms:

The primary mechanism is a retry system with exponential backoff for transient failures. When code execution fails, the agent examines the error message and attempts to reformulate its approach, often breaking down complex analyses into smaller steps or choosing alternative analytical methods.

For more persistent failures, the system employs a "hop" mechanism that allows the agent to abandon the current analytical path and explore alternative approaches. This prevents the agent from becoming stuck in unproductive investigation paths while maintaining the context of what has been learned.

4.4 Specialized Analysis Tools

CodeReAct integrates a comprehensive suite of analysis tools specifically designed for microservice observability data, implemented as composable Python functions for dynamic execution during investigation. We list out the tools provided to the base CodeReAct agent in Appendix B.

4.4.1 Metric Analysis Pipeline

The metric analysis pipeline transforms raw service metrics into actionable insights through multi-stage processing. Beginning with metric ingestion and normalization, it employs clustering techniques to discover patterns across metrics, performs statistical analysis to detect significant changes, and leverages causality testing to identify relationships between metrics. This progression allows the agent to move from raw time series data to understanding metric relationships and system behavior patterns.

4.4.2 Log Analysis Framework

The log analysis framework processes Docker logs to extract and correlate system-level errors across services. Through timestamp parsing, error pattern matching, and temporal analysis, the framework enables identification of error propagation patterns and their temporal relationships across the microservice architecture.

4.4.3 Distributed Trace Analysis

The distributed trace analysis tools transform distributed tracing data into analyzable formats focused on request flows and service interactions. By extracting key performance indicators and analyzing request patterns, these tools enable the agent to understand service dependencies and identify bottlenecks or error propagation paths in request chains.

4.4.4 Generalized Libraries

In addition to domain-specialized tooling, we provide CodeReAct with access to multiple Python libraries and tooling for it to compose its own dynamic actions during inference time. These base tools include Pandas, NumPy, SciPy, and others.

This integrated toolset enables CodeReAct to correlate insights across multiple observability dimensions, building a comprehensive understanding of system behavior during failure scenarios.

4.5 State and Context Management

Rather than maintaining fixed state definitions, CodeReact implements a flexible trajectory tracking system that captures the entire investigation history. This includes:

- Thoughts and reasoning steps taken by the agent
- Executed code and its outputs
- Error states and recovery attempts
- Intermediate findings and hypotheses

This comprehensive state tracking enables the agent to:

1. Reference and build upon previous findings
2. Maintain multiple parallel lines of investigation
3. Backtrack and reexamine data with new context
4. Generate detailed investigation reports that explain its reasoning process

Through this implementation, CodeReact achieves its goal of flexible, iterative root cause analysis while maintaining reproducibility and traceability of its investigation process.

4.6 Implementation Frameworks

For the implementation of the agentic setup, we opt to use DSPy [10]. Our choice to become an early adopter of this framework stems from multiple reasons, including DSPy's relative minimalism compared to other choices, its flexibility, and, most crucially, its prompt optimization faculties that align well with some of our bespoke agent architectures.

Moreover, we are interested in rapidly prototyping and testing sketches without having to worry about manual tuning of prompts to ensure that a given system performs optimally within its design. We believe that this will also enable us to produce more replicable work.

The retriever we utilized in our MSACausalBench evaluation is based on ColBERT V2 [16]. Since ColBERT was trained on natural language rather than code, direct embedding of microservice application code files was not feasible. To address this limitation, we implemented a two-step process: First, we used Claude 3.5 Sonnet to generate synthetic natural language descriptions for each code file, capturing the key functionality, dependencies, and architectural components. These descriptions were then embedded using ColBERT as both the embedder and retriever. We maintained a bidirectional mapping between each description and its corresponding code file, enabling us to retrieve either the semantic description and the actual implementation code during the RCA process.

5 Evaluation

Our evaluation is aimed to assess agentic systems and LLMs to operate on a set of base input artifacts to identify the root cause of SLO violations. Specifically, we restrict our scope to operate within DeathStar Benchmark artifacts running on a single node using Docker and Blueprint [1]. The specific base artifacts which we consider, and that we believe span a comprehensive coverage of the possible artifacts needed to perform RCA are as follows:

- System traces with Jaeger for all user-facing calls.
- Prometheus system performance monitoring time series for all microservices.
- Docker and applications logs.
- Applicaation code.

We evaluate our models on two sets benchmarks as part of the MHQC project: (1) `MSACausalBench`: a curated set of benchmarks built similarly to `HumanEval` [3] aimed to assess agents' ability to perform causal reasoning in the microservice application domain as well as test understanding of the microservice application and its components' subtle dependencies based on the DeathStarBenchmark Hotel; and (2) `SLOvRCABench`: A set of end-to-end failure cases complete with curated artifacts for testing agentic systems on their ability to identify the root causes of faults. In both evaluations, we do not restrict assumptions for a system operating with retrieval augmented generation (RAG) [13] and only focus on reliability and robustness.

We provide our benchmarks in open distribution to help researchers evaluate their work on the SLO violation RCA process. We now discuss these benchmarks in greater detail.

5.1 `MSACausalBench`

The root cause analysis (RCA) process fundamentally relies on causal reasoning capabilities to understand system behavior. Through our experiments with various failure cases, we observe that this reasoning faculty—whether explicitly programmed or implicitly learned through scaling laws—is essential for navigating the RCA process. While there is ongoing debate about LLMs' capacity for novel reasoning [15], we posit that the ability to understand implicit causality within a system (i.e., how different components interact to produce specific outcomes) is crucial for RCA. This understanding enables more effective identification of anomalous behavior and its prerequisite conditions. Furthermore, comprehensive knowledge of application components and their dependencies has proven critical for accurate RCA, leading us to incorporate specific questions about microservice dependencies in our benchmark.

MSACausalBench consists of 112 questions derived from the DeathStar Benchmark (DSB) Hotel application. Each element in the benchmark is structured with:

- An open-ended question testing system understanding
- A golden answer for evaluation
- A category classification

While automated evaluation methods such as BertScore or LLM-based judges could be employed, we opted for manual evaluation to ensure maximum accuracy [19] [25]. The questions span various aspects of system understanding, including:

- Request flow analysis (e.g., "Create a Mermaid sequence diagram showing the flow of method calls from the ReservationHandler function starting from the client, assuming all functions work properly except UserService.")
- Service dependency mapping (e.g., "Which services does GeoService call?")
- Infrastructure understanding (e.g., "If UserService uses a database, which one does it use?")

To evaluate the effectiveness of different retrieval approaches, we tested three distinct RAG configurations:

1. Text descriptions of code files only
2. Combined code and text descriptions
3. Code files only

Each configuration was evaluated to determine the optimal approach for answering questions about the microservice application.

Categories	Description	Code	Both
System Infrastructure	0.88	0.94	0.94
Service Dependencies	1	1	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 1: Proportion of correctly answered questions per category across different input configurations. 'Code' represents using only code files, 'Description' uses only text descriptions, and 'Both' uses both code and descriptions combined.

5.1.1 Results for MSACausalBench

The results demonstrate varying effectiveness across different input configurations for each category of system understanding. System Infrastructure questions showed similar high performance across all configurations (0.88-0.94). Service Dependencies achieved perfect accuracy (1.0) with both code-only and description-only inputs, with a slight decrease (0.94) when combining both. Notably, code-only access significantly outperformed other configurations for Latency Propagation questions (0.88 vs 0.63) and Request Flow analysis (0.88 vs 0.19 for descriptions). This suggests that implementation details are crucial for understanding performance impacts and request patterns. Operational Dependencies benefited most from the combined approach (0.97), indicating that both implementation details and system descriptions contribute to understanding operational requirements. Overall, the code-only configuration demonstrated consistently strong performance across all categories, while description-only inputs showed high variance in effectiveness depending on the question type. The stark performance gap between code-only and description-only approaches, particularly in categories like Request Flow, suggests that significant implementation details are lost in the process of generating natural language descriptions of the code. This information loss highlights the challenges in capturing the full complexity of system behavior in generated descriptions.

5.2 SLOvRCABench

SLO Violation RCA is a broad process that spans many distinct methodologies with their unique input variables and requirements. Consider that works from the literature propose varying methodologies that each operate on different input artifacts to produce insight on anomalies which moves the overall root cause analysis into a simpler domain, including possibly outputting a full description of the root cause. However, we also observe that these disparate procedures operate on either subsets or easy transformations of the base input artifacts we defined.

Drawing on this observation, we provide three end-to-end failure in the open DeathStar Hotel. Specifically, we represent these failure cases as offline timestamped datasets with an input SLO violation description, base artifacts, and an RCA resolution procedure. Our three faults in this case are

1. F1: Hardware level memory of the rate service docker container
2. F2: A network partition of the rate service seperating from the rest of the microservices on the link level
3. F3: A network degradation where 30 percent of packets to rate service are dropped and 70 percent experience an added latency of 100 ms

Failure Case	Correct Analysis	Failed Service	Failed Operatio
F1	2	3	4
F2	0	0	0
F3	2	2	2

Table 2: Performance evaluation across three failure scenarios: F1 (memory stress injection in rate service), F2 (network partition of rate service), and F3 (network degradation with 30% packet loss and 70% packets experiencing 100ms added latency to rate service). "Correct Analysis" indicates the number of times the system provided out of the five experiments we conducted. "Failed Service Identified" shows how many times the system correctly identified the service experiencing the failure. "Failed Operations Identified" represents the number of times the system correctly identified the specific operations that were failing.

Experiment	Successful Halting	SLO Violation Verification
F1	4	5
F2	5	5
F3	5	5

Table 3: Results from running agents on the three failure scenarios: F1 (memory stress in rate service), F2 (network partition of rate service), and F3 (network degradation to rate service). Each failure case was tested 5 times. "Successful Halting" indicates how many times the agent properly concluded its analysis, while "SLO Violation Verification" shows how many times it correctly verified the presence of an SLO violation.

5.3 Results for SLOvRCABench

The results reveal interesting patterns in how the system performs across different types of network failures. For memory stress (F1), the system achieved partial success, correctly identifying the root cause in 2 out of 5 experiments. While its root cause analysis success was moderate, it showed better performance in identifying failed operations (4/5) and the failed service (3/5). The complete network partition (F2) proved to be the most challenging scenario, with the system failing to provide correct analysis or identify either the failed service or operations (0/5 across all metrics). This suggests that total network isolation creates scenarios that are particularly difficult for the system to diagnose. The partial network degradation scenario (F3) showed similar success rates to the memory stress case in terms of correct analysis (2/5), but with more consistent performance across all metrics, achieving 2/5 for both service and operation identification. Notably, while the system struggled with root cause analysis in some cases, it demonstrated strong reliability in SLO violation verification, achieving perfect scores (5/5) across all scenarios. The system also showed good reliability in completing its analysis, with near-perfect halting success (4/5 for F1, 5/5 for F2 and F3).

6 Related Work

6.1 RCA Methods

Traditional RCA approaches can be broadly categorized into graph-based and non-graph-based methods. Graph-based methods rely on various graph structures to model and analyze system behavior. Dependency graphs map component interdependencies and service relationships, enabling fault propagation analysis across complex systems. Bayesian networks extend this by modeling conditional dependencies between system variables, allowing for probabilistic reasoning about failure scenarios.

Non-graph methods have evolved from traditional statistical approaches to more sophisticated machine learning techniques. Statistical approaches analyze correlations between system metrics and performance indicators to identify anomalous behavior patterns. More recently, machine learning methods have emerged as powerful tools for RCA, capable of processing complex, multi-dimensional datasets to predict and identify root causes in large-scale distributed systems [21].

6.2 LLM Based Diagnostic Agents

Recent advancements in cloud service monitoring and root cause analysis (RCA) have leveraged large language models (LLMs) to address the growing complexity of modern microservice architectures. In particular, RCACopilot [4], an automated system for cloud incident RCA, has demonstrated significant improvements in diagnosing faults in large-scale cloud environments. RCACopilot integrates multi-source data (e.g., logs, traces, and metrics) to predict incident root causes and provide engineers with explanatory narratives. The system successfully automates diagnostic data collection and generates hypotheses for root causes, streamlining the incident resolution process for cloud engineers.

Similarly, the MonitorAssistant system [24] further simplifies cloud service monitoring by utilizing LLMs for anomaly detection and alarm interpretation. MonitorAssistant automates the configuration of monitoring models and translates alarms into human-readable reports. It addresses the gap between academic anomaly detection techniques and practical, industry-focused approaches by integrating intuitive, language-based tools for engineers. The system has been deployed successfully in large-scale cloud environments, proving its efficacy in aiding engineers in understanding and resolving service-level violations.

These systems highlight the potential of LLM-based multiagent systems to enhance RCA in complex microservice architectures. Our proposed work builds upon these foundations, integrating reasoning mechanisms such as reflection and search optimization to further improve diagnostic capabilities in industry-standard benchmarks. Additionally, our open benchmarks contribute a novel resource for evaluating intelligent systems in the context of service-level objective (SLO) violation resolution.

7 Discussion

7.1 Takeaways & Limitations

Our results show that CodeReAct is performant and accomplishes the complex tasks of Root Cause Analysis. This is done in both the context of causal reasoning within a specific observation that was resultant from the environment and in the context of composing an action that would transition the current environment. Of those two reasoning archetypes, however, our results and experiments indicate that environment transitions remain the area where greatest improvement is to be seen. This is specifically the case as transitions which involve invocation of tools and/or other interaction with structured data is something that LLMs currently struggle with and ARCA does not attempt to solve.

To fully realize the gains of a perfect transition policy, future work may explore the incorporation of more robust frameworks for interacting with structured data within agents, such as Anthropic's new Model Context Protocol. An orthogonal area of research to overcome the optimal transition limitation is to model transitions more formally using the reinforcement learning paradigm [17]. Specifically, a policy-iteration optimization may be applied to teach the CodeReAct agent to minimize the number of transitions and thus choose the action that is likely to arrive to the root cause in the fewest number of subsequent steps–in other words, to choose the current most insightful action, with exploration and exploitation balanced through a prompt-level monte-carlo tree search (MCTS) [22].

For practical purposes, it is important for us to state that CodeReAct's tooling was significantly limited, as this work approaches the task of RCA from an ML perspective. Successor works may opt to explore the implementation of other agents to synthesize tools during runtime or to use the CodeReAct base agent in a collaborative fashion, where different instances of CodeReAct–coordinated via a router–serve as domain experts through the various tools and signatures made available to them. Finally, agents experiencing coding errors must temporarily change their mode of operation to not enter non-resolving retries.

However, our initial hypothesis that stateful iteration lies at the heart of performance within complex tasks appears to be verified. CodeReAct is, to our knowledge, a first attempt at implementing fully end-to-end LLM agents to perform microservice application root cause analysis.

8 Conclusion

This paper presents ARCA, a novel approach to automating root cause analysis in microservice architectures through LLM-based agents. Our work makes several key contributions to the field: (1) the introduction of CodeReAct, an agent architecture that combines dynamic code execution with flexible state management to mirror the investigative patterns of experienced SREs; (2) the development of two comprehensive benchmarks for evaluating RCA systems; and (3) an open-source toolkit for workload orchestration and fault injection in microservice environments.

Our evaluation results reveal both the promises and limitations of LLM-based approaches to RCA. The `MSACausalBench` results demonstrate strong performance in system understanding tasks, particularly when working directly with code (93% accuracy), suggesting that LLMs can effectively reason about microservice architectures when properly augmented with relevant context. However, the `SLOvRCABench` results highlight ongoing challenges in handling complex failure scenarios, especially in cases of complete network partitions where the system struggled to provide accurate analysis.

Despite current limitations, ARCA represents a significant step toward automated RCA in microservice environments. By providing open benchmarks and tools, we hope to facilitate further research in this critical area of systems engineering. As LLM capabilities continue to evolve, frameworks like CodeReAct that combine computational flexibility with structured reasoning approaches will become increasingly valuable for addressing the challenges of maintaining complex distributed systems.

References

[1] Vaastav Anand, Deepak Garg, Antoine Kaufmann, and Jonathan Mace. Blueprint: A toolchain for highly-reconfigurable microservices. 2023.

[2] Nathan Bronson, Abutalib Aghayev, Aleksey Charapko, and Timothy Zhu. Metastable failures in distributed systems. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 221–227, 2021.

[3] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. 2021.

[4] Yinfang Chen, Huaibing Xie, Minghua Ma, Yu Kang, Xin Gao, Liu Shi, Yunjie Cao, Xuedong Gao, Hao Fan, Ming Wen, et al. Automatic root cause analysis via large language models for cloud incidents. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 674–688, 2024.

[5] Inc Docker. Docker. *lınea].[Junio de 2017]. Disponible en: https://www. docker. com/what-docker*, 2020.

[6] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. Sage: practical and scalable ml-driven performance debugging in microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 135–151, 2021.

[7] Yu Gan, Meghna Pancholi, Dailun Cheng, Siyuan Hu, Yuan He, and Christina Delimitrou. Seer: leveraging big data to navigate the complexity of cloud debugging. In *Proceedings of the 10th USENIX Conference on Hot Topics in Cloud Computing*, pages 13–13, 2018.

[8] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, 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*, pages 3–18, 2019.

[9] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large

language models. *arXiv preprint arXiv:2203.15556*, 2022.

[10] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T Joshi, Hanna Moazam, et al. Dspy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*, 2023.

[11] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213, 2022.

[12] James Lewis and Martin Fowler. Microservices: a definition of this new architectural term. *MartinFowler.com*, 25(14-26):12, 2014.

[13] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020.

[14] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts, 2023.

[15] Iman Mirzadeh, Keivan Alizadeh, Hooman Shahrokhi, Oncel Tuzel, Samy Bengio, and Mehrdad Farajtabar. Gsm-symbolic: Understanding the limitations of mathematical reasoning in large language models. *arXiv preprint arXiv:2410.05229*, 2024.

[16] Keshav Santhanam, Omar Khattab, Jon Saad-Falcon, Christopher Potts, and Matei Zaharia. Colbertv2: Effective and efficient retrieval via lightweight late interaction, 2022.

[17] Richard S Sutton. Reinforcement learning: An introduction. *A Bradford Book*, 2018.

[18] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation. *IEEE Cloud Computing*, 4(5):22–32, 2017.

[19] Aman Singh Thakur, Kartik Choudhary, Venkat Srinik Ramayapally, Sankaran Vaidyanathan, and Dieuwke Hupkes. Judging the judges: Evaluating alignment and vulnerabilities in llms-as-judges, 2024.

[20] Oguzhan Topsakal and Tahir Cetin Akinci. Creating large language model applications utilizing langchain: A primer on developing llm apps fast. In *International Conference on Applied Engineering and Natural Sciences*, volume 1, pages 1050–1056, 2023.

[21] Tingting Wang and Guilin Qi. A comprehensive survey on root cause analysis in (micro) services: Methodologies, challenges, and trends, 2024.

[22] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36, 2024.

[23] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023.

[24] Zhaoyang Yu, Minghua Ma, Chaoyun Zhang, Si Qin, Yu Kang, Chetan Bansal, Saravan Rajmohan, Yingnong Dang, Changhua Pei, Dan Pei, et al. Monitorassistant: Simplifying cloud service monitoring via large language models. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 38–49, 2024.

[25] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. Bertscore: Evaluating text generation with bert, 2020.

[26] Dahlia Ziqi Zhou and Marios Fokaefs. Ai assistants for incident lifecycle in a microservice environment: A systematic literature review, 2024.

A Sample RCA Report Output

Below is a sample output from our RCA system demonstrating the format and depth of analysis provided for a memory-related SLO violation:

```
2024-11-26 23:59:28,639 - main - DEBUG - Starting iteration 8
2024-11-26 23:59:28,639 - main - DEBUG - Formatting trajectory
2024-11-26 23:59:28,639 - main - DEBUG - Getting next thought and action
ROOT CAUSE ANALYSIS REPORT
=========================
ISSUE:
------
SLO Violation: Less than 90% of requests meeting 50ms latency threshold
Current Success Rate: 88.47% (Target: 90%)
ROOT CAUSE:
-----------
The rate service is experiencing performance degradation due to memory management issues, causing cascading timeouts acros
EVIDENCE:
---------
1. Rate Service Metrics:
   - Extremely high memory usage (influence score: ~64.9T)
   - Memory cache issues (primary metric in cluster analysis)
   - Multiple change points with large magnitude changes (~59.4M average)
2. Trace Analysis:
   - Rate service requests show only 81.10% SLO success rate
   - Consistent pattern of ~1.8s delays before rate service calls
   - High error rate (57%) in slowest traces
   - 1,187 RPC deadline exceeded errors in frontend logs
3. Request Flow Pattern:
   - Frontend -> Search -> Rate service chain most affected
   - Rate service calls either timing out or being canceled
   - Consistent ~1.9s latency in worst cases
IMPACT PROPAGATION:
------------------
1. Rate Service: Primary bottleneck with memory issues
2. Search Service: Blocked waiting for rate service responses
3. Frontend: Experiences cascading timeouts
4. End Users: ~11.53% of requests exceed 50ms SLO
RECOMMENDATIONS:
---------------
1. Immediate:
   - Investigate rate service memory management
   - Consider increasing rate service memory limits
   - Implement circuit breaker pattern for rate service calls
2. Short-term:
   - Add request timeouts before 1.8s threshold
   - Implement caching for rate service responses
   - Monitor memory usage patterns
3. Long-term:
   - Review rate service architecture for scalability
   - Implement graceful degradation strategies
   - Add automated memory usage alerts
```

B Tool Suite for the Base CodeReAct Agent

The following appendix contains all tools and imports provided to CodeReAct as part of its analysis procedure. For implemeted tools, we provide the documentation strings. For libraries, we provide the import statements.

```
imports = [
    "import pandas as pd",
    "import numpy as np",
    "from scipy import stats",
    "from tslearn.clustering import KShape",
    "from tslearn.preprocessing import TimeSeriesScalerMeanVariance",
    "from statsmodels.tsa.stattools import grangercausalitytests",
    "import ruptures as rpt",
    "import re",
    "from datetime import datetime",
    "from collections import defaultdict",
    "from pathlib import Path",
    "import json",
]

function_docs = [
    """def read_service_metrics(service_name):
    Reads and processes metrics for a given service, combining time series data with metric descriptors.

    Args:
        service_name (str): Name of the service (without '_service' suffix)

    Returns:
        tuple: A tuple containing:
            - list: List of metric descriptors containing metadata for each metric in the form of a dictionary, with the f
                - "__name__": str,
                - "help": str,
                - "query": str,
                - "service": str,
                - "key": int,
                - "type": counter | gauge
            - pd.DataFrame: DataFrame with timestamps and metric values, with columns renamed according to descriptors
    """,

    """def add_timeseries_to_descriptors(descriptors, metrics_df):
    Adds a time series for each metric to its descriptor dictionary.

    Args:
        descriptors (list): List of metric descriptor dictionaries
        metrics_df (pd.DataFrame): DataFrame containing metrics data with 'timestamp' column and each metric as a column r

    Returns:
        list: Updated descriptors with 'time_series' key containing pd.Series for each metric. The series name is it's __n

    Note:
        Use this function when needing to perform exploratory analysis on metrics. It is more convenient than using the da
    """,

    """def cluster_metrics(descriptors, n_clusters=3):
    Clusters time series metrics using k-shape clustering.

    Args:
        descriptors (list): List of metric descriptors with `time_series` keyes containing pd.Series for each metric
        n_clusters (int): Number of clusters to create

    Returns:
        dict: Mapping of cluster labels to metric names
    """,

    """def summarize_clusters(descriptors, clusters):
```

```
Generates statistical summaries for each cluster of metrics, focusing on detecting sharp changes using ruptures and th

Args:
    descriptors (list): List of metric descriptors with ‘time_series‘ keyes containing pd.Series for each metric
    clusters (dict): Mapping of cluster labels to metric names

Returns:
    dict: Cluster summaries containing statistical information
        num_metrics: Number of metrics in the cluster
        metrics: List of metric names
        mean_std: Average standard deviation across metrics
        mean_range: Average range (max-min) across metrics
        mean_entropy: Average entropy, measuring data distribution complexity
        mean_autocorr: Average autocorrelation, indicating temporal dependencies
        mean_trend: Average linear trend slope
        mean_cv: Average coefficient of variation (std/mean)
        avg_num_change_points: Average number of detected change points
        avg_change_magnitude: Average magnitude of changes at change points
        avg_kl_divergence: Average KL divergence between segments
        avg_spike_count: Average number of spikes (outliers > 3)
""",

"""def identify_primary_metrics(descriptors, clusters, max_lag=5):
Identifies primary metrics within each cluster by analyzing lagged correlation and granger causality relationships.
Primary metrics are determined based on their influence scores, which combine both correlation strength
and granger causality significance.

Args:
    descriptors (list): List of metric descriptors with ‘time_series‘ keys containing pd.Series for each metric
    clusters (dict): Mapping of cluster labels to metric names
    max_lag (int, optional): Maximum lag to consider for correlation and causality analysis. Defaults to 5.

Returns:
    dict: Dictionary containing analysis results for each cluster with the following structure:
        {
            cluster_id: {
                ’primary_metric’: str,  # Name of the metric with highest influence score
                ’influence_scores’: dict,  # Mapping of metrics to their computed influence scores
                ’graph’: dict  # Directed graph representation of metric relationships where each edge contains:
                    - ’target’: Name of the target metric
                    - ’weight’: Edge weight (correlation or causality strength)
            }
        }
""",

"""def read_docker_logs(service_name):
Read docker logs for a specific service.

Reads logs from a file named ’{service_name}_service.log’.
No need to specify path or suffix - these are handled internally.

Args:
    service_name (str): Name of the service to read logs for

Returns:
    list: Lines from the log file if found, empty list otherwise
""",

"""def parse_log_timestamp(log_line):
```

```
Extract and parse timestamp from a log line.

Looks for timestamps in the format [STDERR] YYY/MM/DD HH:MM:SS where YYY is a 3-digit year.
Prepends '2' to the year value since logs show truncated year (e.g. 024 for 2024).

Args:
    log_line (str): The log line to parse

Returns:
    datetime or None: Parsed datetime object if timestamp found and valid, None otherwise
""",

"""def is_system_error(log_line):
Check if a log line contains system-level error patterns.

Searches the log line for common system error patterns like RPC timeouts,
connection refused errors, and connection resets. The search is case-insensitive.

Args:
    log_line (str): The log line to check for system errors

Returns:
    bool: True if any system error pattern is found, False otherwise
""",

"""def analyze_service_logs(logs):
Analyze logs for a single service to identify and summarize system errors.

Processes each log line to detect system errors (like RPC timeouts, connection issues)
and groups them by error type along with their timestamps.

Args:
    logs (list[str]): List of log lines to analyze

Returns:
    defaultdict: A dictionary where:
        - keys are error message strings
        - values are lists of datetime objects representing when each error occurred
""",

"""def generate_error_report(services):
Generate comprehensive error report for all services.

Analyzes logs from multiple services to create a consolidated error report. For each service,
reads Docker logs and identifies system errors, then compiles them into a pandas DataFrame
with error statistics.
No need to specify path or suffix - these are handled internally.

Args:
    services (list[str]): List of service names to analyze

Returns:
    pd.DataFrame: DataFrame containing error report with columns:
        - service: Name of the service where error occurred
        - error_type: Type/description of the error
        - count: Number of occurrences of this error
        - first_seen: Timestamp of first occurrence
        - last_seen: Timestamp of last occurrence
        Returns empty DataFrame if no errors found.
```

```
""",

"""def traces_to_pds(traces):
Convert a list of Jaeger traces into a pandas DataFrame with trace analytics.

Takes a list of Jaeger trace objects and extracts key metrics and attributes into a structured
DataFrame format for analysis. Calculates timing information, error states, and service
relationships for each trace.

Args:
    traces (list[dict]): List of Jaeger trace objects, where each trace contains:
        - traceID: Unique identifier for the trace
        - spans: List of span objects with timing and process information
        - processes: Dictionary mapping process IDs to service information

Returns:
    pd.DataFrame: DataFrame with the following columns:
        - 'trace_id': Unique identifier for the trace
        - 'latency': Total trace duration (latest end time - earliest start time) in microseconds
        - 'total_spans': Number of spans in the trace
        - 'contains_errors': True if any span contains an error tag
        - 'error_rate': The rate of errors encountered in the trace (number of spans with errors / total spans)
        - 'error_messages': Semicolon-separated list of error messages from spans
        - 'operation': Operation name from the root span
        - 'services': Semicolon-separated list of services involved
        - 'longest_span_service': Service name that had the longest self time
        - 'longest_span_duration': Self time of the longest span in microseconds
        - 'mean_span_duration': Average self time of spans in microseconds

Note:
    Root span is identified as the span with no references, or if none found,
    defaults to the span with earliest start time.
"""
,

"""def analyze_jaeger_trace(all_traces, trace_id):
Analyzes a specific Jaeger trace to extract key metrics and information.

This method processes a trace identified by `trace_id` from a collection of traces,
calculating various metrics such as total duration, number of spans, error rate,
and constructing a request flow. It returns a summary of these metrics in a dictionary format.

Args:
    trace_id (str): The unique identifier of the trace to be analyzed.
    all_traces (list): A list of trace dictionaries, each containing trace data.


Returns:
    str: A string representation of a dictionary containing:
        - 'total_duration in microseconds': The total duration of the trace in microseconds.
        - 'total_spans': The total number of spans in the trace.
        - 'error_rate': The rate of errors encountered in the trace.
        - 'error_messages': A dictionary of error messages and their occurrence counts.
        - 'request_flow': A list of dictionaries detailing the request flow,
          including service transitions, operations, start times, and durations.
        - 'longest_span': A dictionary containing information about the span with the longest self-duration.
Raises:
    KeyError: If the trace with the specified `trace_id` is not found in the collection.
""",
    """def load_traces():
    Load Jaeger trace data from a JSON file.

    Returns:
        list: List of traces loaded from the 'data' field of the JSON file
    """
]
```