Leveraging Large Language Models for the Auto-remediation of Microservice Applications: An Experimental Study

Conference Paper · July 2024

DOI: 10.1145/3663529.3663855

Zakeya Namrud
École de Technologie Supérieure

Marin Litoiu
York University

L. Shwartz
T.J Watson Research Center, IBM

Leveraging Large Language Models for the Auto-remediation of Microservice Applications: An Experimental Study

Komal Sarda[†]
York University
Toronto, Canada
komal253@yorku.ca

Zakeya Namrud[†]
York University
Toronto, Canada
Zakeya10@yorku.ca

Marin Litoiu
York University
Toronto, Canada
mlitoiu@yorku.ca

Larisa Shwartz
IBM T. J. Watson Research Center
Yorktown Heights, USA
lshwart@us.ibm.com

Ian Watts
IBM
Toronto, Canada
ifwatts@ca.ibm.com

ABSTRACT

Runtime auto-remediation is crucial for ensuring the reliability and efficiency of distributed systems, especially within complex microservice-based applications. However, the complexity of modern microservice deployments often surpasses the capabilities of traditional manual remediation and existing autonomic computing methods. Our proposed solution harnesses large language models (LLMs) to generate and execute Ansible playbooks automatically to address issues within these complex environments. Ansible playbooks, a widely adopted markup language for IT task automation, facilitate critical actions such as addressing network failures, resource constraints, configuration errors, and application bugs prevalent in managing microservices. We apply in-context learning on pre-trained LLMs using our custom-made Ansible-based remediation dataset, equipping these models to comprehend diverse remediation tasks within microservice environments. Then, these tuned LLMs efficiently generate precise Ansible scripts tailored to specific issues encountered, surpassing current state-of-the-art techniques with high functional correctness (95.45%) and average correctness (98.86%).

CCS CONCEPTS

• **Computing methodologies** → **Natural language processing**; **Artificial intelligence**; • **Information systems** → **Information systems applications**.

KEYWORDS

Ansible, Autonomic computing, Auto-remediation, Cloud native applications, Code generation, Kubernetes, Large language models, Microservices, Prompt engineering, Real-time faults, Self-adaptive software.

ACM Reference Format:
Komal Sarda, Zakeya Namrud, Marin Litoiu, Larisa Shwartz, and Ian Watts. 2024. Leveraging Large Language Models for the Auto-remediation of Microservice Applications: An Experimental Study. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE Companion '24), July 15–19, 2024, Porto de Galinhas, Brazil.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3663529.3663855

1 INTRODUCTION

Companies have shifted from traditional on-premise models to cloud platforms for deploying and managing applications. This transition streamlines deployment and scalability using shared environments and cloud-native technologies [91]. Microservices, akin to service-oriented architectures [12], break down system logic into specialized units, operating within containers to facilitate scalability and foster software reusability [11, 65]. Kubernetes (K8s) [42] is instrumental in managing microservices at scale, offering features like auto-scaling and self-healing [6]. For smaller environments, MicroK8s, a lightweight K8s distribution, is valuable, providing essential components like API server, kubelet, and kubectl [41]. However, this evolution has led to complexities, resulting in production incidents impacting operational systems [87]. For example, during peak shopping periods, Amazon reportedly loses $100 million for every hour of downtime [82]. Despite ongoing efforts in autonomic and adaptive computing [38, 73], many cloud services and applications still experience failures necessitating manual intervention. These failures often occur due to a) the faults and remedies were not considered in the design phase [80] or b) the planning and execution of remediation steps require elaborate workflows using runtime contextual information [13, 88].

Self-adaptation, including self-healing, is meant to autonomously monitor and adjust system behavior to address various stages of incidents [81]. Lately, practitioners and researchers focusing on self-healing have integrated Artificial Intelligence (AI) into IT operations (AIOps) [49], leveraging data-driven AI to automate parts of the incident life-cycle [86] as an industrial solution [34, 69]. This cycle in the industry typically comprises these stages: '***Incident detection and routing***', where incidents are reported either by users or automated monitoring systems, and operators route them to the appropriate teams [64]; '***Root-cause analysis***', involving iterative communication among operators to identify the underlying issue; and '***Remediation execution***', encompassing actions

[†] Both authors contributed equally to this work.

FSE Companion '24, July 15–19, 2024, Porto de Galinhas, Brazil

ACM ISBN 979-8-4007-0658-5/24/07...$15.00
https://doi.org/10.1145/3663529.3663855

taken to restore service functionality [17]. Despite advancements and automation like incident detection and prioritization [57, 72], and automated dependency mapping [52], operators still invest substantial manual effort in writing remediation scripts for incident mitigation steps, involving multiple rounds of communication. [4]. These challenges stem from the sheer number of services and anomalous events, compounded by the absence of predefined remediation solutions and scripts [71]. The extensive scale of cloud-based microservices, exemplified by Netflix's 1000+ microservices [61] and Uber's 4000+ microservices [77], leads to intricate interactions and frequent anomalous events. Moreover, the diversity of these microservices, combined with frequent software/hardware updates, results in identical anomalies manifesting differently across services or distinct anomalies showing similar traits. Manual creation of appropriate remediation scripts for identified root causes becomes a time-consuming, repetitive task, particularly when scripting for similar anomalies with different configurations [38], and requires careful attention to detail. The challenge intensifies as operators must manage unique configuration settings across numerous services amid complex anomalies, making it difficult to write remediation scripts with the necessary variability and ensure compatibility [38]. Even a minor mistake in the remediation script can result in significant discrepancies [35]. Thus, providing a useful remediation script to operators can significantly reduce their effort and expedite the incident mitigation process.

A less explored alternative in AIOps is the use of pre-trained Large Language Models (LLMs) to generate remediation scripts addressing the occurred incident based on the identified root cause, as has been applied in other use cases [9, 79]. LLMs like CodeBert [16], Codex [9], LLaMa [76], GPT-Neo and GPT-NeoX [85], GPT-4 [70] show promise for code generation tasks. While these techniques are well-explored in general-purpose programming languages, their application in domain-specific languages, such as YAML, common in distributed IT platforms and distributed application configuration, has received less attention [66]. YAML files play a crucial role in defining and configuring various aspects of a distributed IT infrastructure, with many companies utilizing Ansible-YAML [20], in conjunction with K8s, to design intelligent, automated responses to root cause alerts in these distributed environments. This approach not only streamlines incident response but also reduces the burden on operators from routine firefighting tasks [24, 62]. Despite advancements in leveraging LLMs to generate Ansible playbooks [66], the focus has primarily been on improving productivity for existing users, emphasizing code completion rather than the creation of entirely new code. There is a pressing need for specialized LLMs tailored for Ansible playbook generation, particularly for auto-remediation, serving as AI assistants for operators [40]. To address challenges related to the cost and maintenance of conventional LLMs, researchers are exploring the few-shot learning capabilities of LLMs [35, 75]. This approach enables incident-specific code generation with minimal examples, eliminating the need for extensive parameter tuning. However, its application has not yet been adapted for Ansible-based code generation tasks, and the efficiency of LLMs in this context remains unknown.

This paper seeks to minimize manual effort in resolving incidents in microservice applications, thereby reducing operators' time spent on searching for, customizing existing, or writing new template scripts. We propose employing LLMs to generate Ansible-based incident remediation code at the distributed system level. An experimental study is conducted to evaluate the effectiveness of in-context tuning on two models: GPT-4 and LLaMa-2-70B. While GPT-4, the largest available LLM is expected to produce good results, it might not be appropriate when prompts include confidential data; on the other hand, open-source models, such as LLaMa-2-70B, can be deployed within the confines of the managed platforms and applications. Our assessment focuses on the accuracy of the generated code, tested through execution on microservice applications setup, and its ability to resolve runtime issues based on standard tasks. We utilize the KubePlaybook dataset [59], dedicated to Ansible playbooks in the context of IT automation and anomaly resolution within cloud-native distributed environments. This dataset is essential for enhancing the few-shot learning capabilities of LLMs, enabling them to autonomously generate more Ansible Playbooks for auto-remediation throughout the incident life cycle. Overcoming these challenges moves us closer to realizing a fully autonomous AIOps environment. **Our research contributions encompass the following key aspects:**

(1) To the best of our knowledge, this study marks the first experimental investigation into the utilization of LLMs for the generation of Ansible playbooks for microservice incident remediation and pioneer the evaluation of LLMs' effectiveness in generating Ansible scripts from natural language (NL) prompts, devoid of any code input.
(2) We evaluate the effectiveness of the LLaMa-2-70B model compared to the GPT-4 model for generating Ansible-based code for on-premise deployment.
(3) The tuned LLM performance is tested on Robot-shop [1], Sock-shop [31] applications as well as on a unseen QOTD [19] sample during tuning within unfamiliar environments like OpenShift [21]. We show that the model achieves a functional correctness rate of up to 95.45% and an average correctness of up to 98.86%.

This study primarily benefits AIOps teams by performing an experimental study and empowering the incident management process in the industry by using IBM industrial infrastructure for automatically recommending or generating Ansible playbooks. The rest of this paper is structured as follows: Section 2 presents the preliminary information and theoretical background. Section 3 outlines our approach and methodology. Section 4 provides an experimental evaluation of our approach and discusses the results. Section 5 addresses practical considerations and identifies potential threats to validity. Section 6 provides a review of relevant literature and discusses works related to our topic. Finally, Section 7 concludes the paper and outlines potential directions for future research.

2 PRELIMINARIES AND PRACTICE

In this section, we describe each phase of the incident life cycle, outlining the processes and advancements within each stage.

2.1 Incident Management and Self-Healing for Microservices

The concept of self-healing entails automatic detection and resolution of issues, thereby enhancing system reliability. Challenges in manual runtime management highlight the need to reconfigure microservices for self-adaptive systems [6]. Consequently, researchers have focused on employing AI-driven incident management approaches for preemptive detection of instance incidents, with the aim of swiftly mitigating potential failures. Incident management involves identifying, diagnosing, and resolving system issues, including predictive strategies [8]. These anomalies may stem from various factors, such as network failures, resource constraints, configuration errors, and application bugs [72, 78]. Operators continually collect metrics and logs, utilizing AI models for automated incident detection and routing. These models automatically detect incidents, utilizing prediction systems to notify operators and guide them to the appropriate teams. While these models effectively identify and alert operators about incidents, traditionally, operators manually address these faults to determine their root causes.

2.2 Fault Localization and Auto-Remediation

The root cause analysis involves delving deeper into the system to pinpoint the fundamental reason behind an incident [48, 78]. This process aids operators in understanding the primary issue triggering the incident, facilitating more effective and targeted solutions. Researchers have explored diverse approaches, from trace-based detection methods [46, 50] to accurately capturing correlated anomalies across various data types, like time series, semi-structured logs, and trace structures [90]. These methodologies aim to enhance incident diagnosis by integrating heterogeneous, dynamically changing data sources like logs\trace\metrics [91]. Recently, LLMs, leveraging extensive historical incident data, have been integrated for root-cause analysis [4].

The incident life cycle has seen advancements in detection, routing, and root cause analysis, yet attention is required in script writing and execution for remediation to achieve autonomy. Auto-remediation techniques automate incident resolution, reducing manual intervention and expediting recovery [26]. The industry's focus on automated server management and cloud elasticity involves microservices in containers for scalability, notably utilizing platforms like K8s, MicroK8s, and OpenShift for self-healing in distributed systems [42]. While K8s integrates autoscaling for self-healing, it lacks infrastructure self-healing capabilities, necessitating additional tools for lower-level issue resolution [23]. OpenShift extends K8s but may encounter limitations in addressing network disruptions or infrastructure problems beyond its scope, potentially leading to incomplete resolutions by focusing solely on immediate symptoms. These platforms and services contribute to alleviating some of these challenges through auto-scaling and recovery features. However, a significant part of the challenges still rely on operational know-how [58]. The arrival of LLMs brings exciting potential to improve and speed up the creation and operation of auto-remediation systems.

2.3 LLMs and Remediation Code Generation

LLMs have found rapid adoption in software engineering to address various challenges [2, 16]. In transformer-based architectures, fundamental to LLMs, each token in the input sequence undergoes encoding into Query (Q), Key (K), and Value (V) vectors, critical for self-attention computations. The Query vector defines the token's context in information search, the Key vector holds pertinent information, and the Value vector supplies crucial content for specific query outputs [22]. However, applying LLMs to specialized tasks like understanding scientific literature or generating code presents challenges due to data discrepancies between their training and evaluation sets in these niche domains [67]. To address this, techniques like unsupervised secondary pre-training aim to bridge this gap, aligning the model's understanding with evaluation data sources [18, 67]. These models leverage NL prompts and task demonstrations to generate code [51]. In this context, prompts play a critical role in directing an LLM to perform specific tasks, such as generating remediation code in response to identified incident causes. Effective prompt engineering, involving detailed and task-specific prompts tailored to factors like configuration, deployment, hosts, and remediation actions, is essential to ensure accurate and practical code generation.

In-Context Learning of LLMs. In-context tuning of LLMs is a technique used to adapt these models to specific tasks without retraining them from scratch or modifying their underlying weights. It involves giving the model a prompt or a set of examples immediately before the task, to guide it on how to respond. In-context tuning leverages Zero-shot, One-shot, and Few-shot learning paradigms to adapt LLMs to new tasks efficiently, utilizing the vast knowledge already embedded in the models from their initial training. Zero-shot learning doesn't provide any example; it relies on the model's generalization from its initial data to entirely new cases. One-shot learning involves in-context tuning the model using just one task-specific example before testing on new data. Few-shot learning broadens the model's learning from a limited set of instances, assessing its ability to apply knowledge to new situations [35].

2.4 Research Questions

Following incident detection and root cause identification, mitigation steps are essential, including actions like code rollback, hotfixes, infrastructure adjustments, or configuration updates. GPT-4 and LLaMa-2-70B, both trained on publicly available data, offer predictive capabilities. While GPT-4 is a proprietary service, providing high performance but raising privacy concerns, LLaMa-2 is an open-source alternative suitable for on-premises deployment, albeit with associated costs. This study explores these models' effectiveness in generating Ansible scripts for addressing production incidents within K8s clusters using the KubePlaybook dataset. Leveraging Ansible for system management, which offers greater scalability compared to traditional kubectl shell commands [24, 56], addresses automation challenges effectively. Therefore, in addition to real-time faults and scrapped code, this study also focuses on using LLMs for kubectl command-based Ansible playbook generation to streamline issue resolution for operators, eliminating the necessity to memorize intricate K8s commands. Through contextual tuning experiments, the objective is to establish an autonomous framework for remediating issues in cloud-native applications. Our work addresses the following original research questions (RQs):

- RQ1: What is the impact of hyperparameters on in-context tuning of LLMs for generating Ansible remediation code in response to NL prompts? In this regard, we conduct experiments with model parameters to identify the optimal hyperparameters tailored to our specific use cases.

- RQ2: How does the volume of training data for different learning techniques affect the efficacy of LLMs in generating Ansible code from NL prompts? We utilize the optimal hyperparameters identified in RQ1 to compare zero-shot, one-shot, and few-shot learning methods, assessing their impacts on Ansible remediation code generation.
- RQ3: How effectively does a few-shot tuned LLM generalize to unseen applications or new deployments? We explore the performance of LLMs with few-shot learning adapted, incorporating best practices identified in RQ1 and RQ2 for real-time fault resolution. This includes their capacity to synthesize platform-specific kubectl commands for K8s clusters and generate playbooks tailored for previously unseen environments such as OpenShift.

3 APPROACH AND METHODOLOGY

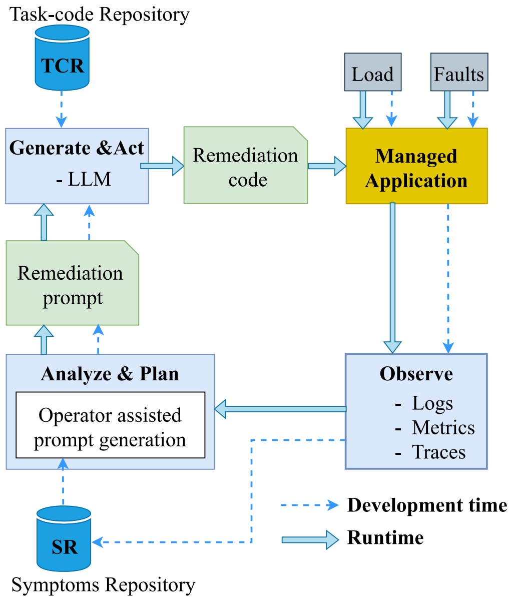


Figure 1: LLM integrated MAPE-k loop architecture

Autonomic management systems involve components such as Monitor, Analyze, Plan, and Execute (MAPE-K) [37]. We integrate LLM-based remediation in MAPE-K loop architecture, specifically supporting the Generate and Act phases of the loop, as illustrated in Figure 1. The loop also includes Observe, Analyze, and Plan services. In our experiments, as Managed Application, we use several sample microservice applications. To monitor microservices effectively and conduct detailed analyses, we utilize industrial observability tools like IBM Instana [29] and IBM Cloud Pak for AIOps [28]. The dashed lines in Figure 1 suggest the data flow prior to deployment of the Managed Application, while the continuous arrow lines suggest data flow at runtime. The architecture uses a task-code repository (*TCR*) aimed at in-context tuning of LLMs for generating Ansible code to remediate runtime anomalies in microservice architectures. **Observe** monitors the metrics, logs, events, traces, and alarms of the application and its environment and stores the data in a Symptoms Repository; **Analyze and Plan** is an operator-assisted activity. An operator will use various tools to identify patterns, detect anomalies and root causes, and devise a remediation plan (Remediation Prompt) in *NL*. It is also possible to generate the remediation prompt by passing the summary of the observed anomaly to the LLM [40]; however, in this paper, we consider the prompt being generated by an operator, as it is in accordance with the state of the practice. **Generate and Act** will convert the *NL* plan into code, using pre-trained GPT-4 or LLaMa-2-70B, and execute it. This component uses an LLM adapted with few-shots learning using data from the *TCR* repository. **Faults** and **Load** are external perturbations for the Managed Application. Prior to deployment time, Faults and Loads are injected to collect training data; in production, they are generated by the application and its environment. The focus of this paper is on the **Generate and Act** component, the automatic generation of Ansible playbooks using LLMs. It includes the *NL* prompt engineering as the input, while the Ansible playbook (remediation code) is the output of **Generate and Act** component.

3.1 Prompt Template for Auto-Remediation

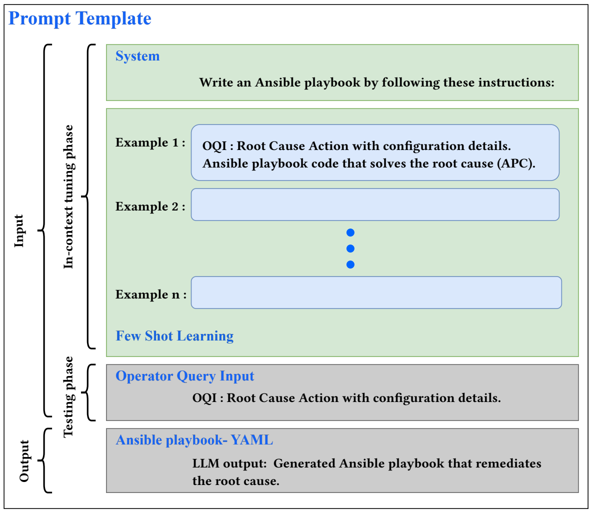

Prompt Template
Input
In-context tuning phase
Testing phase
Output
System
Write an Ansible playbook by following these instructions:
Example 1 :
OQI : Root Cause Action with configuration details.
Ansible playbook code that solves the root cause (APC).
Example 2 :
Example n :
Few Shot Learning
Operator Query Input
OQI : Root Cause Action with configuration details.
Ansible playbook- YAML
LLM output: Generated Ansible playbook that remediates the root cause.

Figure 2: Prompt engineering & few-shot learning

Our primary goals is to craft an effective prompt guiding LLMs to generate precise Ansible playbooks. In Figure 2, the prompt combines three key components crucial for in-context tuning. The initial element, labeled as *System*, provides the model with identity (an essential factor for composing Ansible playbooks based on given directives). This system prompt offers contextual instructions, directing the model's understanding and generation process. It serves as a constant guide, setting predefined parameters for the LLM, focusing solely on generating Ansible playbooks, and ensuring adherence to specific intended tasks and contexts [39, 63]. By defining *System*, we establish the necessary context for the subsequent elements: the Operator Query Input (*OQI*) and the remediation

Ansible Playbook Code (*APC*) example. Each (*OQI*) is crafted with the operator's assistance and encompasses detailed descriptions of the issue, specific conditions, desired outcomes, as well as any constraints or preferences. This approach enables prompts to be tailored precisely to the unique requirements of the IT environment. The prompt, Figure 2, focuses on few-shot learning. Each in-context tuning example includes an (*OQI*) paired with its corresponding (*APC*) solution. During testing, natural language instructions are used as query inputs following the structural conventions set within the (*OQI*) component without (*APC*).

Figure 3 displays an example of (*OQI*). The initial step in prompt construction involves identifying anomalies and their root causes, facilitated by various anomaly detection models [72] and multimodal root-cause models [46, 48]. In complex microservice architectures, specifying precise targets like hostnames, pod names, deployment names, or applications is essential to minimize disruptions during auto-remediation. Our prompt utilizes placeholders such as `<Host_Name>`, `<Name_space>`, and `<Deployment_name>` to simplify management within microservices frameworks. Next, automation actions are defined, guiding LLMs to generate Ansible playbooks based on root-cause actions. This information can originate from root-cause mitigation recommendation models [4] or operators. Additionally, the prompt template serves as a guide for developing prompt templates using LLMs, demonstrating how prompts can be structured for automated Ansible playbook generation within microservices architectures. Placeholders are replaced with actual values, and resulting playbooks are customized for specific microservices environments and deployment details. Prompt engineering evolves based on remediation outcomes. Analyzing Ansible's actions refines prompts, and continuous learning and updates enhance the auto-remediation process. The template assists operators by eliminating the need to write or search for entire code segments, allowing them to focus on reviewing and fixing issues.

3.2 Task-Code Repository (TCR)

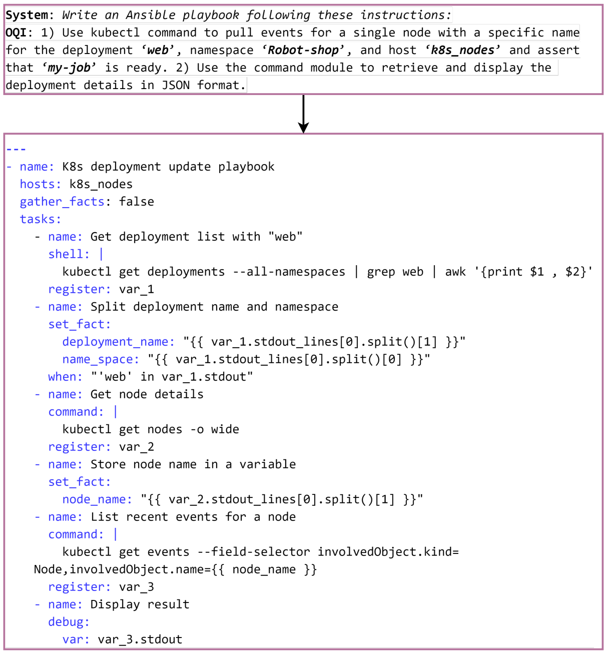
In our experimental study, we utilized a Kubeplaybook repository [59], designed specifically for tuning LLMs. This repository comprises 130 unique Ansible playbooks collected from GitHub and Galaxy, enhanced with kubectl shell commands and real-time fault scenarios. It marks a significant initiative in creating an auto-remediation dataset tailored for LLMs. Currently, all the examples within the repository are Ansible-based YAML files and are designed for Kubernetes environments. For each Ansible playbook code, we create an NL prompt, and we use an example for few-shot learning. The repository encompasses three distinct categories of pairs, comprising Ansible playbook code and corresponding prompts. These categories consist of the following:

(1) **K8s essential commands playbooks:** Among the playbooks in the dataset, 62.3% are primarily designed for essential and common tasks like configuring and deploying in K8s environments. They encompass a wide range of Kubectl shell commands used in K8s operations.

(2) **Scraped Ansible Playbooks from open sources repositories:** This specific portion constitutes 19.2% of our dataset, consisting of Ansible playbooks collected from various repositories on platforms like Galaxy and GitHub. These playbooks encompass a wide array of configurations and deployment scenarios.

(3) **Chaos Engineering Remediation playbooks:** 18.5% of our dataset consists of Ansible playbooks created to handle faults generated through Chaos engineered operational faults [55].

Figure 3 shows a small example from the repository. The Ansible code is aimed at pulling events for a single node with a specific name for the deployment. This particular instance comprises two components: the initial prompt, designated as the *operator input* (*OQI*), and the subsequently generated *Ansible playbook*. By doing so, we aimed to ensure high relevance and specificity in our data, thereby enhancing its potential utility. In essence, this repository serves as a valuable toolkit for automating systems, allowing IT professionals to swiftly migrate from shell commands to Ansible modules to identify, analyze, and resolve runtime anomalies within microservice architectures.



System: *Write an Ansible playbook following these instructions:*
OQI: 1) Use kubectl command to pull events for a single node with a specific name for the deployment ***'web'***, namespace ***'Robot-shop'***, and host ***'k8s_nodes'*** and assert that ***'my-job'*** is ready. 2) Use the command module to retrieve and display the deployment details in JSON format.

```
---
- name: K8s deployment update playbook
  hosts: k8s_nodes
  gather_facts: false
  tasks:
    - name: Get deployment list with "web"
      shell: |
        kubectl get deployments --all-namespaces | grep web | awk '{print $1 , $2}'
      register: var_1
    - name: Split deployment name and namespace
      set_fact:
        deployment_name: "{{ var_1.stdout_lines[0].split()[1] }}"
        name_space: "{{ var_1.stdout_lines[0].split()[0] }}"
      when: "'web' in var_1.stdout"
    - name: Get node details
      command: |
        kubectl get nodes -o wide
      register: var_2
    - name: Store node name in a variable
      set_fact:
        node_name: "{{ var_2.stdout_lines[0].split()[1] }}"
    - name: List recent events for a node
      command: |
        kubectl get events --field-selector involvedObject.kind=
    Node,involvedObject.name={{ node_name }}
      register: var_3
    - name: Display result
      debug:
        var: var_3.stdout
```

Figure 3: An example Ansible playbook generated using LLM with Operator Query Input (OQI).

4 EXPERIMENTAL STUDY

This section explores a detailed experimental investigation that we carried out with the primary goal of assessing the feasibility and effectiveness of our proposed approach to adapt LLMs for Ansible-based remediation code generation. We basically instantiated the architecture presented in Fig. 1 three times: two times on a Kubernetes platform and once on OpenShift platform. For Kubernetes deployment we applied few-shot learning to tune the LLMs using samples from TCR. For OpenShift, we used the models trained for Kubernetes. We focused on distributed system-level Ansible code generation and auto-remediation, leveraging GPT-4 and LLaMa-2-70B. For in-context tuning, we used up to 10 Kubectl commands from our *TCR*, chosen randomly and then we used we used the remaining 120 unseen examples from *TCR* for testing.

4.1 Evaluation Metrics

To assess the effectiveness of our approach, we employed two key performance metrics: Functional Correctness metric (*FC*) [9] and Average Correctness metric (*AC*). The *AC* metric, an evaluation measure we devised, assesses the quality of blocks of code or tasks generated within a single Ansible script compared to the entire script. In Ansible, a task represents a unit of work, such as installing a package, configuring a service, or copying files. Each generated block undergoes testing to determine its success. The effectiveness of these individual tasks directly influences the overall accuracy and correctness of the Ansible playbook. Ansible reports the status of each task individually, indicating whether it was successful, failed, or skipped and based on success count from each playbook we calculated average correctness

$$AC = \frac{\sum_{j=1}^{m} \sum_{i=1}^{n} (C_i / N_i * 100)}{m} * 100 \quad (1)$$

The equation 1 is a mathematical representation used to calculate the *AC* metric by determining the accuracy for each task in an *APC* and then establishing a ratio to indicate the correctness of each *APC* relative to others. The first ratio is obtained by dividing the number of correct tasks (*C*) in each *APC* that passes the test by the total number of tasks (*n*) in that *APC*. Next, the overall ratio is derived by adding up all the individual task accuracy ratios calculated for each *APC*. This sum is then divided by the total number of *APC*s (*m*). It is important to note that, unlike traditional evaluation methods that categorize an entire APC as either correct or incorrect, the *AC* metric adopts a more nuanced approach. The *AC* metric offers a more comprehensive evaluation by considering the accuracy of individual tasks, allowing for a more intricate understanding of the quality of generated code.

Algorithm 1: Functional Correctness

```
Input  : $param\ n$ : total number of playbook.
         $param\ c$ : number of correct playbook.
         $param\ k$ : K in pass@$k$
Output : $pass@k$.
if ($n - c < k$) then
    return 1.0
else
    return $1.0 - np.prod(1.0 - k/np.arange(n - c + 1, n + 1))$
end if
```

We evaluate code functional correctness using the pass@k metric, as referenced in studies by [5, 9, 43]. In this method, a problem is considered solved if at least one out of k code samples passes specified unit tests. Additionally, we're considering the use of an unbiased estimator from [9] demonstrated in Algorithm 1. The algorithm begins by taking three parameters as input: *n* (the total number of playbooks), *c* (the number of correct playbooks), and *k* (the desired number of correct playbooks for pass@*k*). If the difference between the total number of playbooks and the number of correct playbooks ($n - c$) is less than the desired number of correct playbooks (*k*), it implies that even if all the remaining playbooks were correct, the desired success rate would not be achieved. In this case, the algorithm returns a probability of 1.0, indicating a guaranteed success (as it's almost impossible to fall short of the required number of correct playbooks). Otherwise, the algorithm proceeds to calculate the probability of achieving pass@*k* by considering the chances of getting at least *k* correct playbooks out of the remaining ($n - c$) playbooks. It uses a mathematical calculation involving the product of ratios, computed using Numpy functions in this example ($np.prod(1.0 - k/np.arange(n - c + 1, n + 1))$). This calculation helps estimate the probability of achieving the required success rate by considering the chances of getting at least *k* correct playbooks from the remaining ones.

LLM Hyper-Parameters. In adjusting LLM behavior, key parameters like $Temperature$ and $Top - p$ play vital roles. Temperature, within the range of 0 to 1, determines randomness in code generation. Meanwhile, $Top - p$ (ranging from 0 to 1) causes the exclusion of improbable words from the selection pool by choosing the smallest word set whose cumulative probability exceeds *p*. In our approach, we explore various $Temperature$ parameter settings in the context of each model. This investigation aims to understand how different levels of $Temperature$ may influence the model's output. Meanwhile, we keep the $Top - p$ parameter constant, adhering to its default value of 1.0, to maintain control over the study and ensure any changes observed can be attributed to alterations in the $Temperature$ setting. The preference for keeping temperature variable over a static top-p setup stems from the precision, adaptability, and specific control it offers. It allows tuning the output according to the varying demands of coding tasks, optimizing for anything from strict syntax adherence to inventive problem-solving [68].

4.2 RQ1: Is Hyper-parameters Tuning Effective for In-Context Tuning of LLMs?

4.2.1 Approach : In optimizing code generation through in-context tuning, we specifically investigate the impact of the $Temperature$ hyper-parameter on GPT-4 and LLaMa-2-70B models. Our experiments maintain consistency by applying identical conditions for both models, setting a maximum token probability parameter ($Top-p = 1$), and utilizing 10 in-context tuning samples for few-shot learning. Given the token limits of 8,192 for GPT-4 and 4,096 for the LLaMa-2-70B model, implemented using IBM's industrial tool WatsonX [30], we were restricted to using only 10 few-shot learning examples for both models. We opted for the highest possible samples for few-shot learning to achieve maximum accuracy for the task. Although more samples could have been utilized for adapting few-shot learning on GPT-4, we fixed the sample size to 10 for the sake of comparing both models' performance. To evaluate the tuned models, we generate Ansible code for 10 randomly selected prompts from the test code repository (*TCR*), which were not included in the few-shot learning of the model. Our experimentation focuses on pinpointing optimal hyperparameters, with this quantity of examples effectively demonstrating our findings. We employ a few-shot learning technique, varying the $Temperature$ parameter from 0.0 to 1.0, and assess the models using both *FC* and *AC* metrics for comprehensive validation.

Table 1: Assessing different temperature tuning values to enhance LLMs performance.

LLMs	Few-shot Temperature	FC pass@1	AC
LLaMa-2-70B	0.0	30%	66.7%
	0.5	59.9%	81.5 %
	0.6	**70%**	**90.01%**
	0.8	59.9%	80.1% %
	1.0	30%	80.1%
GPT-4	0.0	66.66%	85.65%
	0.5	66.66%	87.04%
	0.6	**88.89%**	**95.83%**
	0.8	77.77%	94.91%%
	1.0	55.55%	81.58%

4.2.2 Results : Our experiments, Table 1, revealed that the choice of *Temperature* significantly impacts model performance during in-context learning. Specifically, setting the *Temperature* to 0.6 yielded optimal results for the models. For instance, the GPT-4 model achieved *FC* and *AC* percentages of 88.89% and 95.83%, respectively, while the LLaMa-2-70B model attained 70% and 90.01%. These outcomes highlight that a *Temperature* of 0.6 encourages models to generate deterministic, precise, and focused outputs. However, adjusting the *Temperature* to 0.8 slightly reduced the models' performance, resulting in *FC* and *AC* percentages of 77.77% and 94.91% for GPT-4 and improved second-place outcomes for LLaMa-2-70B (59.9% *FC* and 81.5% *AC*). Notably, when the Temperature increased to 1.0, there was a significant decline in performance due to increased randomness, negatively impacting model accuracy. This suggests that a *Temperature* value of 0.6 is most effective in guiding models to produce precise, reliable, and concentrated outputs.

Finding from RQ 1: Using an extremely low (0.0) and high (1.0) *Temperature* setting reduced the level of performance and introduced more randomness. Setting it to a moderate *Temperature* (0.6) significantly improves the models' performance.

4.3 RQ2: What is the Best Learning Method?

4.3.1 Approach : In our investigation, we use in-context learning to tune LLaMa-2-70B and GPT-4 models using diverse learning techniques like zero-shot, one-shot, and few-shot learning. Using the same 10 examples used in hyperparameter tuning in RQ1, we adapt this learning method to pre-trained models. We employ *FC* and *AC* methods to verify the performance of LLaMa-2-70B and GPT-4. Our analysis in *RQ*1 found that a *Temperature* value of 0.6 is optimal for few-shot learning, and we extend this tuning to zero-shot and one-shot methods for a more comprehensive evaluation. Additionally, we test the LLMs' capability to generate Ansible code on 10 unseen examples.

Table 2: Evaluation of LLMs on different learning techniques

LLMs	Learning method	FC pass@1	AC
LLaMa-2-70B	Zero-Shot	0 %	0 %
	One-Shot	30%	40%
	Few-Shot	**70%**	**90.01%**
GPT-4	Zero-Shot	12.50 %	27.08%
	One-Shot	31.25%	62.50%
	Few-Shot	**88.89%**	**95.83%**

4.3.2 Results : The analysis in *RQ*1 led to the determination of an optimal *Temperature* value of T=0.6 for our task. This setting was fixed for all experiments. Table 2 presents the performance comparison of zero-shot, one-shot, and few-shot learning methods. In the zero-shot scenario, LLaMa-2-70B encountered challenges, showing 0% accuracy in both *FCpass@*1 and *AC* metrics. On the other hand, GPT-4 exhibited better performance, achieving 12.50% *FC* and 27.08% *AC* accuracy. Despite GPT-4's relatively superior performance, the low accuracy underscores the significant difficulty both models face in generating precise code without specific in-context tuning. In a single training example scenario, both models show notable improvements in code generation accuracy. The *FC* score increases to 30% for LLaMa-2-70B and 31.25% for GPT-4, with even more substantial gains in *AC*, reaching 40% for LLaMa-2-70B and 62.50% for GPT-4. These results underscore the positive impact of even a single example on enhancing the models' performance. Moving to a few-shot evaluation using ten training examples, both models demonstrate impressive progress. LLaMa-2-70B achieves *FC* and *AC* scores of 70% and 90.01%, while GPT-4 excels further, reaching 88.89% and 95.83%, respectively. The model exhibits impressive accuracy in generating Ansible code for Kubectl commands, even with minimal training examples, showcasing the effectiveness of few-shot learning. GPT-4 outperforms LLaMa-2-70B across various learning methods, particularly excelling in few-shot learning. In-context tuning adjusts the model's parameters for Q, K, and V vectors to fit specific datasets. Limited data prompts more focused adjustments, refining these vectors for intricate task understanding while preserving the model's base knowledge. This adaptive process helps focus attention on vital input details, balancing broader comprehension with task-specific nuances. Swiftly adapting Q, K, and V values tailors model outputs effectively, maintaining a balance between generalization and tuning accuracy [25].

Finding from RQ 2: LLaMa-2-70B and GPT-4 struggle with zero-shot approaches, but their accuracy improves notably with just one sample. Comparing up to 10 examples significantly boosts LLMs adaptation to task-specific code generation. This enhancement leads to 95.83% accuracy for GPT-4 and 90.01% for LLaMa-2-70B.

4.4 RQ3: How Generalizable are In-Context Tuned LLM Models?

With established optimal hyper-parameters (T=0.6) and the identified learning method (Few-shot) for both LLMs, we have standardized these settings for further analysis. Our primary objective is to evaluate the models' capabilities in generating Ansible remediation code for both Kubectl commands and real-time faults. This involves an expanded testing set involving 71 Kubectl commands and 24 real-time faults. Additionally, we assess whether LLMs, tuned on the Kubectl environment with sample microservice applications like "Robot-shop" and "Sock-shop," can adapt to new, unseen environments beyond their few-shot learning examples. This investigation is pivotal in understanding the models' adaptability and generalization to handle diverse, unforeseen scenarios. For both objectives, we maintain consistency with the ten few-shot examples from previous experiments, ensuring a reliable benchmark for evaluating model performance.

In-context tuned LLMs: Real-Time Fault Resolution & Kubectl Command Execution

4.4.1 Approach : Utilizing few-shot learning with GPT-4 and LLaMa-2-70B, we tested the model's ability to generate playbooks for 71 kubectl commands and 24 real-time faults, including DNS errors, DNS faults, Node I/O stress, Pod API latency, Overrides header values of API requests, Node memory hog, Resources overload, etc. These specific faults were chosen due to the absence of a comprehensive industrial real-time dataset. In our future research, we plan to gather a more diverse industrial dataset to further assess the capabilities of our tuned LLMs. The experimental conditions are uniform, with a fixed $Temperature$ parameter (T=0.6) and a maximum token probability parameter (Top-p=1).

Table 3: Evaluation of LLaMa-2-70B & GPT-4 for real time faults & Kubectl commands

Use-case	LLMs	FC pass@1	AC
Real-time fault	LLaMa-2-70B GPT-4	62.5 % **88%**	75% **92.36%**
Kubectl commands	LLaMa-2-70B GPT-4	77.27 % **95.45%**	95% **98.86%**

4.4.2 Results : In an evaluation of GPT-4 and LLaMa-2-70B models using 71 new kubectl command examples from *TCR*, Table 3 outlines their performance metrics. GPT-4 exhibited a high *FC* rate of 95.45% and an *AC* rate of 98.86%, while LLaMa-2-70B showed a slightly lower *FC* rate of 77.27% and an *AC* rate of 95%. Moreover, when tested on 24 real-time faults, GPT-4 delivered an *FC* rate of 88% and an *AC* rate of 92.36%. In comparison, LLaMa-2-70B displayed a lower *FC* rate of 62.5% and an *AC* rate of 75%.

Adaptability of in-context tuned LLMs for Unseen Environments

4.4.3 Approach : We tested LLMs' adaptability by generating playbooks for new, unseen environments like OpenShift and applications such as QOTD. Our evaluation involved designing prompts specific to these environments and generating Ansible code using tuned models. These unexplored environments may introduce different characteristics and shell commands compared to our training data. To assess the performance of LLaMa-2-70B and GPT-4, we manually executed the generated playbooks.

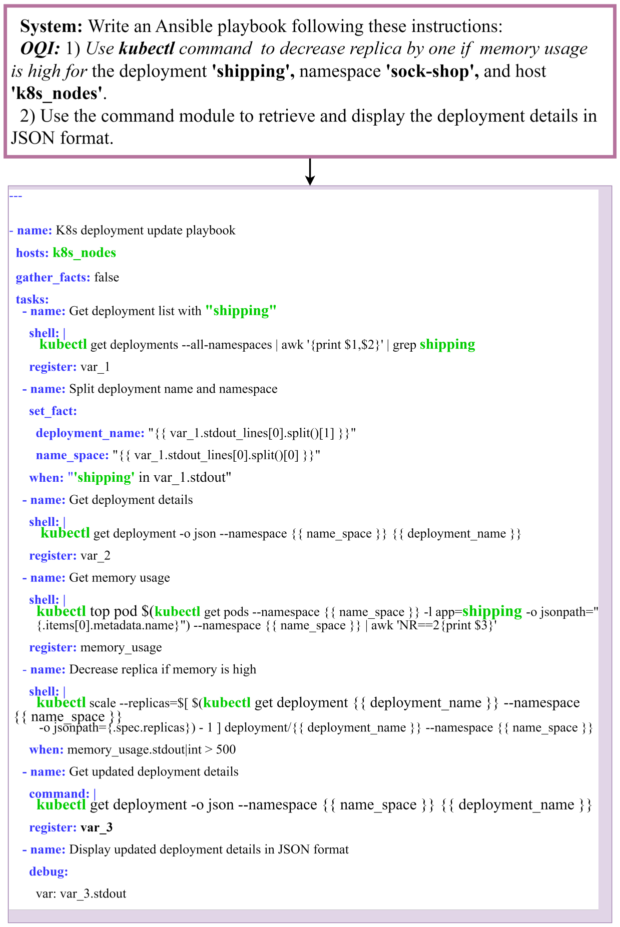
4.4.4 Results : Both models demonstrated robust overall performance in these uncharted territories. In Figure 4, we showcase an exemplary *APC* generated by tuned LLMs. This playbook addresses the issue of resource overload in a Kubectl environment, specifically targeting the host "K8s_nodes" and the application "Robot-shop". Importantly, the same prompt was utilized across different environments, hosts, and applications, illustrating the models' adaptability and versatility.

> **Finding from RQ 3:** GPT-4 achieves the highest accuracy of 92.36% in addressing real-time faults, whereas LLaMa-2-70B achieves a top accuracy of 75%. GPT-4 & LLaMa-2-70B tuned based on K8s environment (kubectl), it is able to generate *APC* for the new, unseen environment (OpenShift) using new OC shell command.

Our findings revealed that LLMs possess the capability to seamlessly adapt to new environments, such as *OpenShift*, different applications (*QOTD*) and (*Sock-Shop*), and diverse hosts (*localhost*). This adaptability was achieved by leveraging *OC* shell commands for obtaining environment information to rectify faults, as demonstrated in Figure 5 and Figure 4. To validate the reliability of the generated Ansible playbook, rigorous manual testing was conducted, resulting in the successful resolution of identified issues. This compelling evidence underscores the ability of tuned LLMs, especially when trained with few-shot techniques, to generalize effectively across various environments, applications, and hosts.

5 DISCUSSION & THREATS

A. Practical Considerations: The experiments shows that, with limited in-context training, general purpose pre-trained LLMs can achieve good performance for microservice auto-remediation tasks. Some production environments contain confidential information in logs, traces and events. From our industrial collaborators, we know that production environments will not send data or prompts to public LLMs. LLMs (such as LLaMa-2) that can run on the same premises as the managed application can alleviate such concerns. Engaging IT operators in root-cause analysis, prompt adjustments, and Ansible code validation minimizes risk and maximizes trust. As confidence in the system grows, operators will increasingly trust automated code generation, speeding up incident resolution. LLMs require a few shot tuning to fit specific platforms and remediation methods. We found that integrating Chaos engineering into DevOps can help tune LLMs in-context to achieve the desired results. A large public dataset with real-world examples will also help fine-tune LLMs for broader use. It will also enable lexical and semantic validation of remediation code [44]. ***B. Threats to Validity***: We evaluated Ansible code generation by two LLMs deployed on industry-scale infrastructures, namely GPT-4 as public service and LLaMa-2-70B as on-premise. LLaMa-2-70B was installed on our industrial partner premises. Other LLMs can yield varying results and accuracies, however, our focus was on industrial-scale models. Additionally, we developed a specific pattern for developing prompts using operator inputs, acknowledging that different prompt styles might yield varied results based on industry types. The evaluation

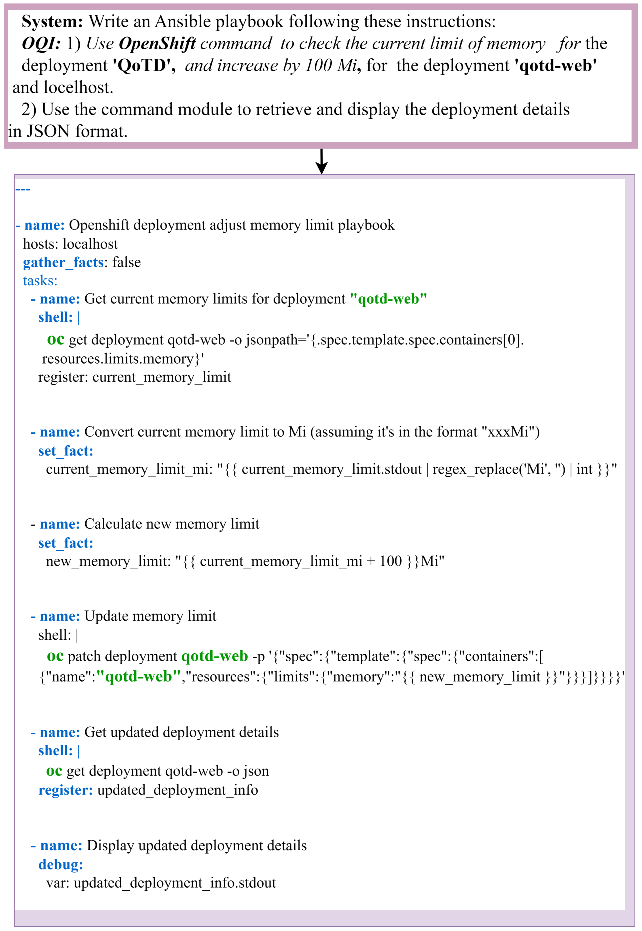


System: Write an Ansible playbook following these instructions:
OQI: 1) *Use **kubectl** command to decrease replica by one if memory usage is high for* the deployment **'shipping',** namespace **'sock-shop',** and host **'k8s_nodes'**.
2) Use the command module to retrieve and display the deployment details in JSON format.

```
---

- name: K8s deployment update playbook
 hosts: k8s_nodes
 gather_facts: false
 tasks:
  - name: Get deployment list with "shipping"
    shell: |
     kubectl get deployments --all-namespaces | awk '{print $1,$2}' | grep shipping
    register: var_1
  - name: Split deployment name and namespace
    set_fact:
     deployment_name: "{{ var_1.stdout_lines[0].split()[1] }}"
     name_space: "{{ var_1.stdout_lines[0].split()[0] }}"
    when: "'shipping' in var_1.stdout"
  - name: Get deployment details
    shell: |
     kubectl get deployment -o json --namespace {{ name_space }} {{ deployment_name }}
    register: var_2
  - name: Get memory usage
    shell: |
     kubectl top pod $(kubectl get pods --namespace {{ name_space }} -l app=shipping -o jsonpath="
     {.items[0].metadata.name}") --namespace {{ name_space }} | awk 'NR==2{print $3}'
    register: memory_usage
  - name: Decrease replica if memory is high
    shell: |
     kubectl scale --replicas=$[ $(kubectl get deployment {{ deployment_name }} --namespace
{{ name_space }}
      -o jsonpath={.spec.replicas}) - 1 ] deployment/{{ deployment_name }} --namespace {{ name_space }}
    when: memory_usage.stdout|int > 500
  - name: Get updated deployment details
    command: |
     kubectl get deployment -o json --namespace {{ name_space }} {{ deployment_name }}
    register: var_3
  - name: Display updated deployment details in JSON format
    debug:
     var: var_3.stdout
```

Figure 4: Ansible playbook to remediate resource overload for K8s environment and Sock-shop app.



System: Write an Ansible playbook following these instructions:
OQI: 1) *Use **OpenShift** command to check the current limit of memory for* the deployment **'QoTD',** *and increase by 100 Mi***,** for the deployment **'qotd-web'** and locelhost.
2) Use the command module to retrieve and display the deployment details in JSON format.

```
---

- name: Openshift deployment adjust memory limit playbook
  hosts: localhost
  gather_facts: false
  tasks:
   - name: Get current memory limits for deployment "qotd-web"
     shell: |
      oc get deployment qotd-web -o jsonpath='{.spec.template.spec.containers[0].
     resources.limits.memory}'
     register: current_memory_limit


   - name: Convert current memory limit to Mi (assuming it's in the format "xxxMi")
     set_fact:
      current_memory_limit_mi: "{{ current_memory_limit.stdout | regex_replace('Mi', '') | int }}"


   - name: Calculate new memory limit
     set_fact:
      new_memory_limit: "{{ current_memory_limit_mi + 100 }}Mi"


   - name: Update memory limit
     shell: |
      oc patch deployment qotd-web -p '{"spec":{"template":{"spec":{"containers":[
     {"name":"qotd-web","resources":{"limits":{"memory":"{{ new_memory_limit }}"}}}]}}}}'


   - name: Get updated deployment details
     shell: |
      oc get deployment qotd-web -o json
     register: updated_deployment_info


   - name: Display updated deployment details
     debug:
      var: updated_deployment_info.stdout
```

Figure 5: Ansible playbook to remediate resource overload for Openshfit environment and QOTD app.

involved testing the prompt construction and Ansible code generation on e-commerce applications, such as Robot-shop, QOTD, and Sock-shop using Kubernetes and OpenShift. These results might not apply to other classes of applications or platforms. Lastly, the remediation code adopted a specific pattern aimed at extracting deployments, implementing changes based on the remediation solution, and displaying the results.

6 RELATED WORK

We categorize the related work into three main topics:

6.1 Available Datasets for Auto-Remediation

Regarding auto-remediation, the availability of datasets depends on the specific use case or problem being explored. While logs, metrics, and alerts can be readily available for training anomaly detection models, collecting data specifically for the code generation for the auto-remediation process may require additional considerations. Notably, HumanEval [9], explored Python code-writing abilities using publicly available GitHub code of the GPT language model. Other relevant datasets include CodeXGLUE [54], and CodeSearchNet [27], which focuses on code generation and professional annotations. Additionally, data-driven techniques and LLMs have been employed for automating aspects of the occurrence lifecycle, as seen in 20-MAD [10]. Other researchers, like [4], have utilized internal datasets containing root causes and mitigating factors. However, there is a requirement for a specialized Ansible repository with specific prompts for each code, facilitating the tuning of LLMs. While the Andromeda dataset introduced by [62] offers valuable insights into the Ansible Galaxy ecosystem, it lacks individual prompts for roles and necessitates additional processing to be compatible with LLM in-context tuning. In this research, we have proposed a dataset with prompts and relevant Ansible codes for Kubectl commands and real-time faults.

6.2 Auto-Remediation Frameworks for Microservices

Auto-remediation techniques in the realm of IT and system management have evolved significantly, introducing innovative methods to automate recovery strategies and streamline resolution processes for diverse operational scenarios. [89] developed the MET framework for bug detection in conflict-free replicated data types, streamlining the process. However, manual intervention is needed for

Table 4: Comparing our work to existing auto-remediation efforts.

Ref	Building Dataset	LLMs Learning	Code generating	Ansible playbook	Performance	Scope
[4]	◑	●	○	○	86.78%	Auto-remediation of root cause analysis
[60]	○	●	●	○	89.31%	Bug fixing, code completion and assertion generation
[45]	○	●	●	○	85.39%	Sketch-based code generation to mimic developers' code reuse behavior
[9]	◔	●	●	○	77.5%	Explore the LLMs and their abilities to write Python code
[47]	●	◔	○	○	33%	utilizing ML to predict the remedies for unknown failures
[14]	●	●	◕	○	50%	Automated program repair
Our Work	●	●	●	●	98.86%	Auto-remediation of root cause analysis

bug-fixing. [32] used Seq2Seq for automated recovery commands, but manual intervention might be required for command execution, especially in complex systems. In [83], the authors presented a system that identifies reasons for performance drops in microservices. It pinpoints problematic services causing issues and detects unusual metrics related to those services. The method involves finding problematic services using a service dependency graph and then flagging abnormal service metrics using an autoencoder based on their reconstruction errors. However, Their metric localization method can only pinpoint the root cause when there's a noticeable deviation from normal values. In [26], authors introduced WRS, which is a recommendation system designed to assist operators in constructing remediation rules. [84] proposed MicroRAS, an automatic recovery method for microservice systems that mitigates performance issues without relying on historical failure data, achieving significant improvements in recovery effectiveness and reduced interference compared to baseline methods. However, MicroRAS is a single-step mitigation approach. [53] developed RESIN, a solution to address memory leaks in large production cloud systems. RESIN utilizes a low-overhead detection method, live heap snapshots, and automated mitigation to identify and resolve memory leaks. In comparison, our approach aims to provide remediation for unknown anomalies at the distributed system level. Notable autonomous incident management for cloud services is done by [4]. They demonstrated the usefulness of SOTA LLMs for mitigating production incidents using internal Microsoft incident data and performed a human study to prove the approach's efficacy. IBM and Redhat collaborated on a research project [66] that introduced Ansible Wisdom. This tool translates natural language into Ansible-YAML code using LLMs, with a focus on boosting IT automation efficiency, particularly in resolving cloud-related issues. The above tool developed primarily emphasizes code completion rather than generating entirely new code.

6.3 LLMs for Code Generation

Numerous efforts have been made in Software Engineering to tackle various complex problems using LLMs. One of these notable applications is code-generation tools like Github Copilot, which leverages GPT-3 to generate code automatically from natural language inputs [9]. Moreover, numerous scholars have conducted research on code generation [9, 85], creating docstrings [3, 9], and repairing code [15, 36]. [33] proposes enhancing LLMs by incorporating post-processing steps based on program analysis and synthesis techniques, resulting in improved performance. Therefore, [7] demonstrated the effectiveness of few-shot learning in various tasks, including code mutation, generating test oracles from natural language documentation, and creating test cases. [60] researchers applied LLMs to enhance several areas of programming, including program repair, assertion generation, code completion, and bug fixing. This application proved instrumental in elevating both developer productivity and the overall quality of the code. Researchers have also drawn inspiration from developers' code reuse practices, leading to the SKCODER, a sketch-based code generation method [45]. Additionally, [74] demonstrated code generation abilities in LLMs. However, this study distinguishes itself by employing advanced LLMs to automate the generation of *APC* for the purpose of conducting root cause analysis auto-remediation. To the best of our knowledge, this research represents a novel endeavor in the field. Table 4 comprehensively compares our study and existing literature. It effectively encapsulates the contrast in methodologies, techniques, and outcomes, particularly in the context of several key parameters: dataset creation, utilization of LLMs, code generation strategies, Ansible playbooks, and the scope of these elements.

7 CONCLUSIONS AND FUTURE WORK

This paper represents the inaugural experimental study into employing LLMs for the creation of Ansible playbooks aimed at incident remediation, setting the stage for assessing LLMs' efficacy in generating Ansible scripts directly from NL prompts, without any code input. Our approach introduces a novel method for auto-remediation within microservices, utilizing LLMs and Ansible playbooks. LLMs generate customized Ansible playbook tasks for microservices by processing prompts structured based on operator-guided root-cause analysis. Leveraging the open-source KubePlaybook repository, once tuned within the specific context, LLMs dynamically analyze the captured context and produce Ansible playbook code suitable for automatic execution. This method provides greater adaptability compared to traditional approaches reliant on inflexible, pre-defined remediation strategies. Our work lays the groundwork for advancing the entire MAPE loop. Future strides involve broadening automating prompt construction through event, log, and metric analysis to encompass elements like severity levels, historical data, predictive resource usage, and dependency-based deployment identification. These enhancements will guide our next steps, ensuring a more robust remediation process. Moreover, by identifying failure patterns and integrating a feedback loop, we aim to enhance the effectiveness of our remediation efforts continually. Also, we plan to enhance the current architecture by incorporating a second LLM layer to achieve prompt construction.

REFERENCES

[1] 1steveww et al. 2023. *Robot Shop is a sample microservice application.* Retrieved Mar 6, 2023 from https://github.com/instana/robot-shop

[2] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333* (2021).

[3] Toufique Ahmed and Premkumar Devanbu. 2022. Few-shot training LLMs for project-specific code-summarization. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–5.

[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. *arXiv preprint arXiv:2301.03797* (2023).

[5] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).

[6] Meriem Azaiez and Walid Chainbi. 2016. A multi-agent system architecture for self-healing cloud infrastructure. In *Proceedings of the International Conference on Internet of things and Cloud Computing*. 1–6.

[7] Patrick Bareiß, Beatriz Souza, Marcelo d'Amorim, and Michael Pradel. 2022. Code generation tools (almost) for free? a study of few-shot, pre-trained language models on code. *arXiv preprint arXiv:2206.01335* (2022).

[8] Jian Chen, Fagui Liu, Jun Jiang, Guoxiang Zhong, Dishi Xu, Zhuanglun Tan, and Shangsong Shi. 2023. TraceGra: A trace-based anomaly detection for microservice using graph deep learning. *Computer Communications* 204 (2023), 109–117.

[9] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[10] Maëlick Claes and Mika V Mäntylä. 2020. 20-MAD: 20 years of issues and commits of Mozilla and Apache development. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 503–507.

[11] Leonardo Duenas-Osorio and Srivishnu Mohan Vemuru. 2009. Cascading failures in complex infrastructure systems. *Structural safety* 31, 2 (2009), 157–167.

[12] Mark Endrei, Jenny Ang, Ali Arsanjani, Sook Chua, Philippe Comte, Pål Krogdahl, Min Luo, and Tony Newling. 2004. *Patterns: service-oriented architecture and web services.* IBM Corporation, International Technical Support Organization New York, NY ....

[13] Conserving Energy. 2006. Transactions on Autonomous and Adaptive Systems. (2006).

[14] Zhiyu Fan, Xiang Gao, Abhik Roychoudhury, and Shin Hwei Tan. 2022. Automated Repair of Programs from Large Language Models. *arXiv preprint arXiv:2205.10583* (2022).

[15] Zhiyu Fan, Xiang Gao, Abhik Roychoudhury, and Shin Hwei Tan. 2022. Improving automatically generated code from Codex via Automated Program Repair. *arXiv preprint arXiv:2205.10583* (2022).

[16] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[17] Jaafar Gaber. 2011. Action Selection Algorithms for Autonomous System in Pervasive Environment: A Computational Approach. *ACM Trans. Auton. Adapt. Syst.* 6, 1, Article 10 (feb 2011), 6 pages. https://doi.org/10.1145/1921641.1921651

[18] Suchin Gururangan, Ana Marasović, Swabha Swayamdipta, Kyle Lo, Iz Beltagy, Doug Downey, and Noah A Smith. 2020. Don't stop pretraining: Adapt language models to domains and tasks. *arXiv preprint arXiv:2004.10964* (2020).

[19] Red Hat. 2023. *QoTD.* Retrieved Oct 11, 2023 from https://github.com/redhat-developer-demos/qotd.git

[20] Red Hat. 2023. *Red Hat Ansible Automation Platform.* Retrieved Nov 27, 2023 from https://www.redhat.com/en/technologies/management/ansible

[21] Red Hat. 2023. *Red Hat OpenShift Container Platform.* Retrieved Oct 11, 2023 from https://www.redhat.com/en/technologies/cloud-computing/openshift/container-platform

[22] Adrián Hernández and José M Amigó. 2021. Attention mechanisms and their applications to complex systems. *Entropy* 23, 3 (2021), 283.

[23] Anouar Hilali, Hatim Hafiddi, and Zineb El Akkaoui. 2021. Microservices Adaptation using Machine Learning: A Systematic Mapping Study. *ICSOFT* (2021), 521–532.

[24] Eric Horton and Chris Parnin. 2022. Dozer: Migrating shell commands to Ansible modules via execution profiling and synthesis. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*. 147–148.

[25] Zhiqiang Hu, Yihuai Lan, Lei Wang, Wanyu Xu, Ee-Peng Lim, Roy Ka-Wei Lee, Lidong Bing, and Soujanya Poria. 2023. LLM-Adapters: An Adapter Family for Parameter-Efficient Fine-Tuning of Large Language Models. *arXiv preprint arXiv:2304.01933* (2023).

[26] Hongyi Huang, Wenfei Wu, and Shimin Tao. 2022. WRS: Workflow Retrieval System for Cloud Automatic Remediation. In *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 1–10.

[27] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

[28] IBM. 2023. *cloud-pak-for-aiops.* Retrieved Nov 21, 2023 from https://www.ibm.com/products/cloud-pak-for-aiops/

[29] IBM. 2023. *instana.* Retrieved Oct 23, 2023 from https://www.ibm.com/products/instana

[30] IBM. 2023. *watsonx-ai.* Retrieved Sep 24, 2023 from https://www.ibm.com/products/watsonx-ai

[31] idcrosby et al. 2017. *Sock Shop : A Microservice Demo Application.* Retrieved Aug 17, 2017 from https://github.com/helidon-sockshop/sockshop

[32] Hiroki Ikeuchi, Akio Watanabe, Tsutomu Hirao, Makoto Morishita, Masaaki Nishino, Yoichi Matsuo, and Keishiro Watanabe. 2020. Recovery command generation towards automatic recovery in ict systems by seq2seq learning. In *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 1–6.

[33] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2022. Jigsaw: Large language models meet program synthesis. In *Proceedings of the 44th International Conference on Software Engineering*. 1219–1231.

[34] Jiajun Jiang, Weihai Lu, Junjie Chen, Qingwei Lin, Pu Zhao, Yu Kang, Hongyu Zhang, Yingfei Xiong, Feng Gao, Zhangwei Xu, et al. 2020. How to mitigate the incident? an effective troubleshooting guide recommendation technique for online service systems. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1410–1420.

[35] Yuxuan Jiang, Chaoyun Zhang, Shilin He, Zhihao Yang, Minghua Ma, Si Qin, Yu Kang, Yingnong Dang, Saravan Rajmohan, Qingwei Lin, et al. 2023. Xpert: Empowering Incident Management with Query Recommendations via Large Language Models. *arXiv preprint arXiv:2312.11988* (2023).

[36] Harshit Joshi, José Cambronero Sanchez, Sumit Gulwani, Vu Le, Gust Verbruggen, and Ivan Radiček. 2023. Repair is nearly generation: Multilingual program repair with llms. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 5131–5140.

[37] Jeffrey O Kephart and David M Chess. 2003. The vision of autonomic computing. *Computer* 36, 1 (2003), 41–50.

[38] Cornel Klein, Reiner Schmid, Christian Leuxner, Wassiou Sitou, and Bernd Spanfelner. 2008. A survey of context adaptation in autonomic computing. In *Fourth International Conference on Autonomic and Autonomous Systems (ICAS'08)*. IEEE, 106–111.

[39] Ryno Kleinhans. 2023. *Safeguard your LLM against user prompt-related shenanigans.* Retrieved Dec 05, 2023 from https://www.linkedin.com/pulse/safeguard-your-llm-against-user-prompt-related-shenanigans-nzeyf/i

[40] Sarda Komal, Namrud Zakeya, Rouf Raphael, Ahuja Harit, Rasolroveicy Mohammadreza, Litoiu Marin, Shwartz Larisa, and Watts Ian. 2023. ADARMA Auto-Detection and Auto-Remediation of Microservice Anomalies by Leveraging Large Language Models. In *Proceedings of the 33rd Annual International Conference on Computer Science and Software Engineering*. 200–205.

[41] Heiko Koziolek and Nafise Eskandani. 2023. Lightweight Kubernetes Distributions: A Performance Comparison of MicroK8s, k3s, k0s, and Microshift. In *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering*. 17–29.

[42] Kubernetes. 2024. *Kubernetes.* Retrieved Jan 30, 2024 from https://kubernetes.io/

[43] Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. 2019. Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems* 32 (2019).

[44] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. 2019. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461* (2019).

[45] Jia Li, Yongmin Li, Ge Li, Zhi Jin, Yiyang Hao, and Xing Hu. 2023. Skcoder: A sketch-based approach for automatic code generation. *arXiv preprint arXiv:2302.06144* (2023).

[46] Zeyan Li, Junjie Chen, Rui Jiao, Nengwen Zhao, Zhijun Wang, Shuwei Zhang, Yanjun Wu, Long Jiang, Leiqin Yan, Zikai Wang, et al. 2021. Practical root cause localization for microservice systems via trace analysis. In *2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS)*. IEEE, 1–10.

[47] Fred Lin, Antonio Davoli, Imran Akbar, Sukumar Kalmanje, Leandro Silva, John Stamford, Yanai Golany, Jim Piazza, and Sriram Sankar. 2020. Predicting remediations for hardware failures in large-scale datacenters. In *2020 50th Annual IEEE-IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S)*. IEEE, 13–16.

[48] Fred Lin, Keyur Muzumdar, Nikolay Pavlovich Laptev, Mihai-Valentin Curelea, Seunghak Lee, and Sriram Sankar. 2020. Fast dimensional analysis for root cause investigation in a large-scale service environment. *Proceedings of the ACM on*

Measurement and Analysis of Computing Systems 4, 2 (2020), 1–23.

[49] Marin Litoiu, Ian Watts, and Joe Wigglesworth. 2021. The 13th cascon workshop on cloud computing: engineering aiops. In *Proceedings of the 31st Annual International Conference on Computer Science and Software Engineering*. 280–281.

[50] Ping Liu, Haowen Xu, Qianyu Ouyang, Rui Jiao, Zhekang Chen, Shenglin Zhang, Jiahai Yang, Linlin Mo, Jice Zeng, Wenman Xue, et al. 2020. Unsupervised detection of microservice trace anomalies through service-level deep bayesian networks. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 48–58.

[51] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *Comput. Surveys* 55, 9 (2023), 1–35.

[52] Yushan Liu, Xiaokui Shu, Yixin Sun, Jiyong Jang, and Prateek Mittal. 2022. RAPID: Real-Time Alert Investigation with Context-aware Prioritization for Efficient Threat Discovery. In *Proceedings of the 38th Annual Computer Security Applications Conference*. 827–840.

[53] Chang Lou, Cong Chen, Peng Huang, Yingnong Dang, Si Qin, Xinsheng Yang, Xukun Li, Qingwei Lin, and Murali Chintalapati. 2022. {RESIN}: A Holistic Service for Dealing with Memory Leaks in Production Cloud Infrastructure. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 109–125.

[54] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664* (2021).

[55] Sehrish Malik, Moeen Ali Naqvi, and Leon Moonen. 2023. CHESS: A Framework for Evaluation of Self-adaptive Systems based on Chaos Engineering. *arXiv preprint arXiv:2303.07283* (2023).

[56] Pavel Masek, Martin Stusek, Jan Krejci, Krystof Zeman, Jiri Pokorny, and Marek Kudlacek. 2018. Unleashing full potential of ansible framework: University labs administration. In *2018 22nd conference of open innovations association (FRUCT)*. IEEE, 144–150.

[57] Yoichi Matsuo and Daisuke Ikegami. 2021. Performance analysis of Anomaly Detection Methods for Application System on Kubernetes with auto-scaling and self-healing. In *2021 17th International Conference on Network and Service Management (CNSM)*. IEEE, 464–472.

[58] Xu Mingyang, Liu Ryan, Tahvildari Ladan, and Stoodley Mark. 2023. An Educational Course on Self-Adaptive Systems using IBM Technologies. In *Proceedings of the 33rd Annual International Conference on Computer Science and Software Engineering*. 143–148.

[59] Zakeya Namrud, Komal Sarda, Marin Litoiu, Larisa Shwartz, and Ian Watts. 2024. KubePlaybook: A Repository of Ansible Playbooks for Kubernetes Auto-Remediation with LLMs. In *Companion of the 15th ACM/SPEC International Conference on Performance Engineering* (<conf-loc>, <city>London</city>, <country>United Kingdom</country>, </conf-loc>) *(ICPE '24 Companion)*. Association for Computing Machinery, New York, NY, USA, 57–61. https://doi.org/10.1145/3629527.3653665

[60] Noor Nashid, Mifta Sintaha, and Ali Mesbah. 2023. Retrieval-based prompt selection for code-related few-shot learning. In *Proceedings of the 45th International Conference on Software Engineering (ICSE'23)*.

[61] Netflix. 2023. *the-story-of-netflix-and-microservices*. Retrieved Nov 21, 2023 from https://www.geeksforgeeks.org/the-story-of-netflix-and-microservices//

[62] Ruben Opdebeeck, Ahmed Zerouali, and Coen De Roover. 2021. Andromeda: A dataset of Ansible Galaxy roles and their evolution. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 580–584.

[63] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[64] Raja Parasuraman, Mustapha Mouloua, Robert Molloy, and Brian Hilburn. 2018. Monitoring of automated systems. In *Automation and human performance*. CRC Press, 91–115.

[65] Gunjan Pathak and Monika Singh. 2023. A Review of Cloud Microservices Architecture for Modern Applications. In *2023 World Conference on Communication & Computing (WCONF)*. IEEE, 1–7.

[66] Saurabh Pujar, Luca Buratti, Xiaojie Guo, Nicolas Dupuis, Burn Lewis, Sahil Suneja, Atin Sood, Ganesh Nalawade, Matt Jones, Alessandro Morari, et al. 2023. Automated Code generation for Information Technology Tasks in YAML through Large Language Models. *arXiv preprint arXiv:2305.02783* (2023).

[67] Alan Ramponi and Barbara Plank. 2020. Neural unsupervised domain adaptation in NLP—a survey. *arXiv preprint arXiv:2006.00632* (2020).

[68] Fernando Vallecillos Ruiz, Anastasiia Grishina, Max Hort, and Leon Moonen. 2024. A Novel Approach for Automatic Program Repair using Round-Trip Translation with Large Language Models. *arXiv preprint arXiv:2401.07994* (2024).

[69] Amrita Saha and Steven CH Hoi. 2022. Mining root cause knowledge from cloud service incident investigations for AIOps. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*. 197–206.

[70] Katharine Sanderson. 2023. GPT-4 is here: what scientists think. *Nature* 615, 7954 (2023), 773.

[71] Bento R. Siqueira, Fabiano C. Ferrari, and Rogério De Lemos. 2023. Design and Evaluation of Controllers based on Microservices. In *2023 IEEE/ACM 18th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 13–24. https://doi.org/10.1109/SEAMS59076.2023.00013

[72] Jacopo Soldani and Antonio Brogi. 2022. Anomaly detection and failure root cause analysis in (micro) service-based cloud applications: A survey. *ACM Computing Surveys (CSUR)* 55, 3 (2022), 1–39.

[73] Roy Sterritt, Manish Parashar, Huaglory Tianfield, and Rainer Unland. 2005. A concise introduction to autonomic computing. *Advanced engineering informatics* 19, 3 (2005), 181–187.

[74] Shailja Thakur, Baleegh Ahmad, Zhenxing Fan, Hammond Pearce, Benjamin Tan, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. 2023. Benchmarking Large Language Models for Automated Verilog RTL Code Generation. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1–6.

[75] Catherine Tony, Markus Mutas, Nicolás E Díaz Ferreyra, and Riccardo Scandariato. 2023. LLMSecEval: A Dataset of Natural Language Prompts for Security Evaluations. *arXiv preprint arXiv:2303.09384* (2023).

[76] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[77] Uber. 2023. *How Uber is monitoring 4,000 microservices with its open sourced Prometheus platform*. Retrieved Nov 21, 2023 from https://www.cncf.io/case-studies/uber/

[78] Muhammad Waseem, Peng Liang, Mojtaba Shahin, Amleto Di Salle, and Gastón Márquez. 2021. Design, monitoring, and testing of microservices systems: The practitioners' perspective. *Journal of Systems and Software* 182 (2021), 111061.

[79] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. 2022. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903* (2022).

[80] Danny Weyns, Ilias Gerostathopoulos, Nadeem Abbas, Jesper Andersson, Stefan Biffl, Premek Brada, Tomas Bures, Amleto Di Salle, Matthias Galster, Patricia Lago, Grace Lewis, Marin Litoiu, Angelika Musil, Juergen Musil, Panos Patros, and Patrizio Pelliccione. 2023. Self-Adaptation in Industry: A Survey. *ACM Trans. Auton. Adapt. Syst.* 18, 2, Article 5 (may 2023), 44 pages. https://doi.org/10.1145/3589227

[81] Danny Weyns, Bradley Schmerl, Masako Kishida, Alberto Leva, Marin Litoiu, Necmiye Ozay, Colin Paterson, and Kenji Tei. 2021. Towards better adaptive systems by combining mape, control theory, and machine learning. In *2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 217–223.

[82] S Wolfe. 2018. "Amazom's one hour of downtime on prime a day may have cost it up to $100 million in lost sales,". In *https://www.businessinsider.com/amazon-prime-day-website-issues-cost-it-millions-in-lost-sales-2018-7*. businessinsider.

[83] Li Wu, Jasmin Bogatinovski, Sasho Nedelkoski, Johan Tordsson, and Odej Kao. 2020. Performance diagnosis in cloud microservices using deep learning. In *International Conference on Service-Oriented Computing*. Springer, 85–96.

[84] Li Wu, Johan Tordsson, Alexander Acker, and Odej Kao. 2020. MicroRAS: Automatic Recovery in the Absence of Historical Failure Data for Microservice Systems. In *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*. IEEE, 227–236.

[85] Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. 1–10.

[86] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. 2016. Early detection of configuration errors to reduce failure damage. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 619–634.

[87] Shuai Yuan, Sanjukta Das, Ram Ramesh, and Chunming Qiao. 2018. Service agreement trifecta: Backup resources, price and penalty in the availability-aware cloud. *Information Systems Research* 29, 4 (2018), 947–964.

[88] Farzin Zaker, Marin Litoiu, and Mark Shtern. 2022. Formally Verified Scalable Look Ahead Planning For Cloud Resource Management. *ACM Trans. Auton. Adapt. Syst.* 17, 3–4, Article 6 (dec 2022), 23 pages. https://doi.org/10.1145/3555315

[89] Yuqi Zhang, Yu Huang, Hengfeng Wei, and Xiaoxing Ma. 2023. Model-checking-driven explorative testing of CRDT designs and implementations. *Journal of Software: Evolution and Process* (2023), e2555.

[90] Chenyu Zhao, Minghua Ma, Zhenyu Zhong, Shenglin Zhang, Zhiyuan Tan, Xiao Xiong, LuLu Yu, Jiayi Feng, Yongqian Sun, Yuzhi Zhang, et al. 2023. Robust Multimodal Failure Detection for Microservice Systems. *arXiv preprint arXiv:2305.18985* (2023).

[91] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. 2018. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering* 47, 2 (2018), 243–260.

Received 2024-02-08; accepted 2024-04-18