Exploring LLM-based Automated Repairing of Ansible Script in Edge-Cloud Infrastructures

Sunjae Kwon[1], Sungu Lee[1], Taehyoun Kim[1], Duksan Ryu[2,*] and Jongmoon Baik[1]

[1]*Korea Advanced Institute of Science and Technology, Daejeon, Republic of Korea*
[2]*Jeonbuk National University, Jeonju, Republic of Korea*
E-mail: duksan.ryu@jbnu.ac.kr
[*]*Corresponding Author*

Received 07 August 2023; Accepted 29 September 2023;
Publication 23 December 2023

Abstract

Edge-Cloud system requires massive infrastructures located in closer to the user to minimize latencies in handling Big data. Ansible is one of the most popular Infrastructure as Code (IaC) tools crucial for deploying these infrastructures of the Edge-cloud system. However, Ansible also consists of code, and its code quality is critical in ensuring the delivery of high-quality services within the Edge-Cloud system. On the other hand, the Large Langue Model (LLM) has performed remarkably on various Software Engineering (SE) tasks in recent years. One such task is Automated Program Repairing (APR), where LLMs assist developers in proposing code fixes for identified bugs. Nevertheless, prior studies in LLM-based APR have predominantly concentrated on widely used programming languages (PL), such as Java and C, and there has yet to be an attempt to apply it to Ansible. Hence, we explore the applicability of LLM-based APR on Ansible. We assess LLMs' performance (ChatGPT and Bard) on 58 Ansible script revision cases from Open Source Software (OSS). Our findings reveal promising prospects, with LLMs generating helpful responses in 70% of the sampled cases. Nonetheless, further research is necessary to harness this approach's potential fully.

Keywords: Edge-cloud, Ansible, Bard, large langue model, automated program repairing.

1 Introduction

The Edge-Cloud system is a crucial computing infrastructure designed to enhance response times for processing Big data through extensively distributed infrastructures [1, 4]. This low-latency responsiveness is critical in advancing modern society, particularly in smart cities and factories. Nevertheless, the manual deployment of such massive infrastructures is fraught with errors and consumes significant time. Infrastructure-as-Code (IaC) [17, 19] is a vital tool empowering developers to efficiently provision and manage the Edge-Cloud system's massive infrastructures using reusable code instead of manual labor. Ansible is the most popular IaC tool owing to its straightforward functionality and a framework implemented using Python and YAML, languages familiar to various users. However, given that Ansible is a collection of code, its code quality directly correlates with the quality of service within the Edge-Cloud system. Consequently, various studies have been dedicated to ensuring the correctness of Ansible, thereby assuring service quality [5, 7, 10, 22, 24, 31].

Concurrently, Automated Program Repairing (APR) has emerged as a dynamic field among various Software Engineering (SE) tasks. APR seeks to automatically furnish potential code patches when confronted with buggy code, offering developers a means to reduce both effort and time spent on bug fixes significantly, consequently curtailing software development costs [18, 24, 25]. Recently, APR researchers have actively applied the Large Language Model (LLM), a deep learning model pre-trained on billions of text and code tokens, due to its remarkable performance on various SE tasks [2, 9, 16, 20, 28]. Furthermore, the LLM-based APRs have demonstrated better performance than state-of-the-art APR techniques [8, 24].

Nevertheless, prior LLM-based APR studies evaluated their performance on widely used Programming Languages (PL), such as Java and C [8, 21, 24, 26, 29, 30]; Ansible has yet to be studied. In the case of Ansible, compared to Java and C, the availability of training data is notably limited, potentially impacting the reliability of findings from prior studies. Therefore, we explore the applicability of LLM-based APR on Ansible through experiments. We evaluate the performance of the two version of ChatGPT and Bard, which are the most actively used LLMs, on 58 Ansible script revision cases from Open Source Software (OSS) repositories enrolled in GitHub. Our findings indicate that the three models provide helpful responses to solve the cases (i.e., offer valuable insights to address cases) in 41 of 58 cases. However, the responses provided by individual models remained relatively modest, underscoring the imperative for further research to establish a practical framework for real-world program repair.

Our contributions are as follows: (1) We evaluate the LLM-based ARP performance on Ansible script for the first time. (2) We curate defective Ansible scripts from OSS enrolled in GitHub to confirm the applicability of LLM-based ARP. (3) We conduct comparative assessments involving three LLMs, exploring a variety of prompts and defect types, with the aim of elucidating strategies for making prompts to effectively guide LLMs in generating error-free Ansible scripts.

2 Background and Related Work

2.1 Edge-Cloud System and IaC

Figure 1 shows the overall structure of Edge-Cloud system. The main advantage of Edge-Cloud system is collecting and processing Big data with low latency using more infrastructures, i.e., Edge nodes in Figure 1, geographically located close to the user or data source [1, 4]. Such locating Edge nodes support the implementation of the Internet of Things (IoT) or Internet

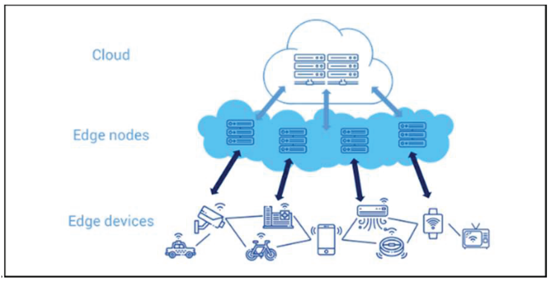
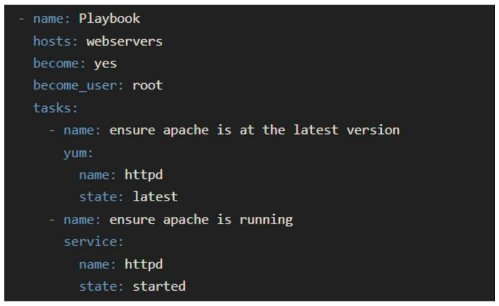


Figure 1 Overall structure of Edge-Cloud system.[1]

[1]https://www.alibabacloud.com/en/knowledge/what-is-edge-computing.



```
- name: Playbook
  hosts: webservers
  become: yes
  become_user: root
  tasks:
    - name: ensure apache is at the latest version
      yum:
        name: httpd
        state: latest
    - name: ensure apache is running
      service:
        name: httpd
        state: started
```

Figure 2 Ansible script example for deploying a webserver.[2]

of Everything (IoE), which is challenging to implement within traditional Cloud systems [1, 4].

Nevertheless, the manual implementation and management of the massive infrastructure required for the Edge-Cloud system can be highly error-prone and time-consuming. IaC is a concept designed to automate these laborious and error-prone tasks by leveraging reusable code [5, 7, 19]. Thanks to the advantages of IaC, it is widely employed in deploying Cloud infrastructure, and the same applies to Edge-Cloud systems.

2.2 Ansible

Ansible is one of the most popular IaC tools [5, 6, 17]. Originating as open-source software (OSS), Ansible has evolved into an end-to-end automation platform provided by Red Hat, configuring systems, deploying software, and orchestrating complex workflow. Figure 2 shows a simple Ansible script example of deploying a webserver. This script employs YAML formatting, with the entire script referred to as a Playbook. Within each Playbook, multiple tasks are defined, each commencing with the '*-name*' tag, which succinctly outlines the task's purpose in natural language. Although not expressed in the script, each task is executed by internally invoking the corresponding Python code. Consequently, these tasks are executed sequentially in the order of the script.

While Ansible offers substantial conveniences, it is important to recognize that, like any other software, Ansible is not immune to bugs. These bugs can significantly impact the overall quality of the system. Bugs in Ansible have been associated with a range of Cloud service outages [5, 7, 22, 25, 32].

[2]https://www.middlewareinventory.com/blog/ansible-playbook-example/.

Researchers have conducted studies focused on enhancing the quality of Ansible scripts to mitigate the risk of such outages. Opdebeeck et al. [22] employ graph algorithms to identify problematic variables within the script, referred to as "*smelly variables*." Dalla Palma et al. [5] predict scripts that are prone using static analysis results from Ansible scripts. Meanwhile, Kwon et al. [15] confirm that LLM can identify the Ansible script and recommend an error-free version of the Ansible script.

We extend Kwon et al.'s study by comparing two prominent Large Language Models (LLMs), ChatGPT and Bard, with various prompt patterns and issues. We aim to find the optimal prompt to guide LLMs in resolving issues on the Ansible script.

2.3 LLM-based APR

Large Language Model (LLM) is a deep learning model pre-trained on billions of text and code tokens, and it generates the most relevant next word to a given sequence of words. LLM's capabilities have significantly advanced with the expansion of training data and the increase in model parameters [3, 24]. Recently, LLMs have been fine-tuned by using Reinforcement Learning from Human Feedback (RLHF) to respond to a wide range of user queries, and the representative examples are Bard[3] and ChatGPT.[4] While LLMs are typically utilized for general purposes, such as Natural Language Processing (NLP), their powerful capabilities have also been found in automating SE tasks [2, 9, 14, 16, 28]. Tian et al. [27] demonstrate that ChatGPT can provide developers valuable insights for resolving software bugs. Xia et al. [29] utilize the conversational manner of LLM for patch generation and validation. Jin et al. [8] combine the capabilities of LLM with a static analyzer. It is an end-to-end program repair framework that identifies source code bugs to execute the necessary repairs.

Despite the ongoing research efforts in the field, the prior studies have concentrated on Java and C, which are widely used in software development. However, research on Ansible has yet to be done. Since the information on Ansible was expected to be relatively scarce compared to C and Java when learning the model, there exists the potential for discrepancies in the results obtained from previous studies. Consequently, our objective is to evaluate the automated repairing performance of LLMs on Ansible.

[3]https://bard.google.com/
[4]https://chat.openai.com/

3 Approach

3.1 GitHub Pull-Request (GHPR)

We gathered cases to evaluate the performance of Large Language Models (LLMs) in Automated Program Repair (ARP) by utilizing the GitHub Pull-Request (GHPR) workflow, which is a standard process on GitHub for addressing issues in developing software. Figure 3 provides an overview of the GHPR process. Initially, when a software issue arises, a developer creates a separate branch from the main branch, which represents the original code thread of a project, to address the issue. After resolving the issue, the developer submits a Pull Request (PR) to the maintainers of the main repository, requesting a review of the modified code. If the maintainers approve the PR, the changes are merged into the main branch. All GHPR-related information is stored on GitHub, and this data can be accessed through the GitHub API. Due to the ease of collecting information using the GitHub API, several studies have utilized it to gather data for various Software Engineering (SE) research tasks [12, 13, 15, 16, 31].

We gathered cases to evaluate the performance of LLMs in APR by utilizing GitHub Pull-Request (GHPR), one of GitHub's workflows, to resolve issues on the developing Software. Figure 2 provides the overall process of GHPR. Initially, when an issue arises, a developer creates a separate branch from the main branch, the original thread of code for a project, to resolve the issue. After addressing the issue, the developer submits a Pull Request (PR) to the main repository maintainers requesting a review of the modified code. If the maintainers accept the PR, the changed code is merged into the main branch. All GHPR-related information is stored on GitHub, and this data can be accessed through the GitHub API.[5] Due to the ease of collecting information using the GitHub API, several studies have utilized it to gather data for various Software Engineering (SE) research tasks [12, 13, 15, 16, 31].

We implemented a Python-based tool for crawling the information using the API. This tool enabled us to collect three distinct types of information: the original code (i.e., Pre-modified code), issue information (i.e., Symptoms of the issue), the changed code to address the issue (i.e., Post-modified code). We consider the changes merged into the main branch reliable, as the repository's maintainers have reviewed and approved these modifications. Consequently, we employ the merged code as the ground truth for our evaluation process.

[5]https://docs.github.com/en/rest?apiVersion=2022-11-28

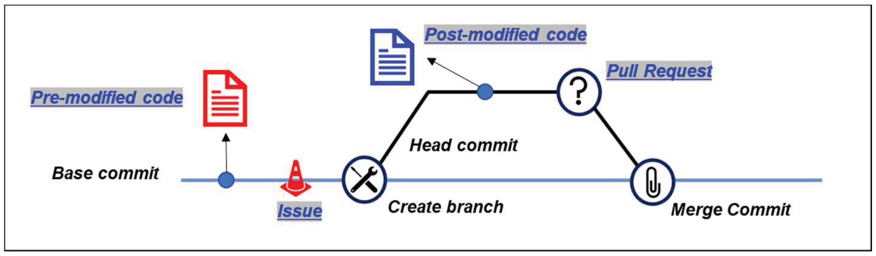


Figure 3 Overview of the GHPR process.

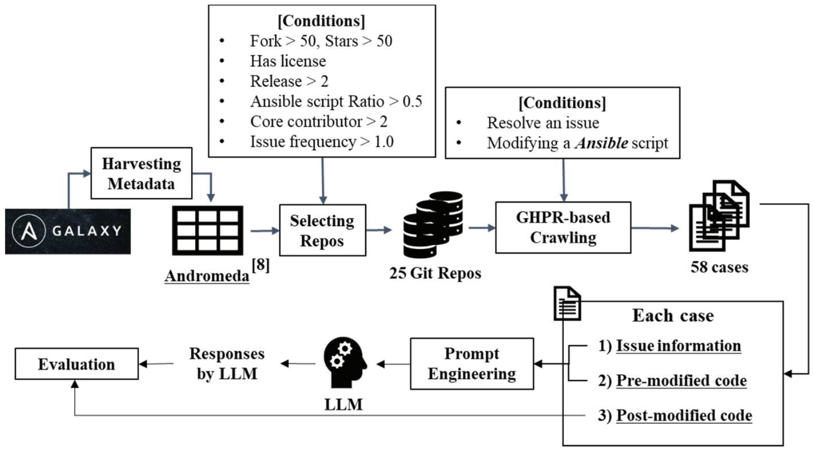


Figure 4 Overall process for collecting cases and evaluation LLM.

3.2 Collecting Cases and Evaluation

Figure 4 provides an overview of the process involved in data-collecting cases and evaluating LLM using the collected cases. The first line represents the data-collecting process, and the second line represents the evaluating process. The data-collecting process begins with the Andromeda dataset [21] containing metadata of approximately 25K Ansible projects enrolled in Ansible Galaxy,[6] a platform for sharing pre-packaged Ansible works. We applied six criteria to select the necessary repositories. Initially, we chose repositories with (1) more than 50 forks and stars and (2) licenses to filter out unserious projects (i.e., homework or personal purposes). Second, we gathered projects that (3) consist of over half of Ansible, (4) more than two contributors, and (5) releases. We also confirmed that the (6) issue frequency is more than

[6]https://galaxy.ansible.com/

one per month to select actively developing Ansible projects. Consequently, we selected 25 Ansible repositories actively developing and serious projects using the criteria. Utilizing our self-implemented crawling tool based on GHPR, as mentioned in Section 3.1, we collected 61 cases, each resolving an issue by modifying an Ansible script.

The evaluation process commences using the collected cases comprising issue information, pre-modified, and post-modified code. The issue information provides a natural language description of the issue's symptoms. The pre-modified code represents the Ansible script afflicted with the issue, and the post-modified code is the human-modified code to resolve the issue. Subsequently, we generate prompts that include an instruction for LLM to propose issue-free Ansible scripts based on the issue information and pre-modified code from each case. Finally, we assess the APR performance of the LLM by comparing the script suggested by the LLM in response to the prompts and the human-modified script, which is post-modified code.

4 Experimental Setup

4.1 Dataset

Table 1 presents an overview of the 58 collected cases, categorized into three distinct types. The first type is '*Modification/Addition*,' which entails cases where the issues can be resolved by modifying or introducing new code. Out of the 58 cases, 30 were addressed by modifying the code, while 20 required adding new lines. The second type, '*Single-line/Multi-line*,' distinguishes between cases where an issue can be resolved by modifying or adding just a single line of code and cases where multiple lines of code need to be altered or added. The last type, '*Single-task/Multi-task*,' relates to the structure of Ansible scripts, which consist of several independent tasks. Thus, the necessary modifications or additions to address an issue can occur within a single task or span multiple tasks. This classification allows us to explore the performance of LLM-based APR on Ansible from various angles, considering the diversity of issues and the scope of changes required for resolution.

4.2 Research Questions

This study establishes 2 research questions and they are as follows.

(1) RQ1: Does prompt affect LLM based ARPs' performance? (Analysis on the pattern of the prompt)

Table 1 Characteristics of the 58 collected cases

Case Type	# of Case
Modification/Addition	30/28
Single-line/Multi-line	19/39
Single-task/Multi-task	35/23
Total	58

A prompt is a sequence of tokens we provide to the LLMs, and these models generate their responses based on the information provided in the prompt. Prior studies show that LLMs' outputs are sensitive to the given prompt. Therefore, it is crucial to carefully engineer prompts to ensure that LLMs produce the desired responses with our intentions. Thus, we began by following the prompt engineering guidelines provided by OpenAI.[7] These guidelines offer specific formats for prompts that align more effectively with user intent and tend to yield better results. The first instruction in the guideline is "*Put instructions at the beginning of the prompt and use ### or """ to separate the instruction and context*." We construct our prompt framework following these instructions. Subsequently, we crafted a variety of prompts by incrementally adding more context to each one. Each case in our study consisted of issue information, including an issue title and body. We generate a variance of prompts by adding these one by one, and the prompts for our experiment are shown in Table 2.

Prompt1 provides the LLM with the defective Ansible script and makes it fix it. This prompt is designed to ascertain the LLM's ability to comprehend the given Ansible script, and accurately predict and fix an issue within the script. Prompt2 additionally provides LLM an issue title, which is the implicit information regarding the issue in the defective Ansible script. We designed this prompt to examine how the performance of LLMs is affected when they are given supplementary implicit information about the issue. Similarly, Prmopt3 additionally provides LLM with an issue body, which offers detailed information on the issues. The issue body includes the symptoms, the precise location of bugs within the code, and the desired direction for fixing them. Using the third prompt, we aim to analyze how LLMs perform when presented with a wealth of context related to specific issues.

RQ2: Does different LLM show similar performance? (Analysis on the type of the LLMs)

[7]https://help.openai.com/en/articles/6654000-best-practices-for-prompt-engineering-with-openai-api

Table 2 Variance of prompts

Type	Prompt
Prompt1	*You will be provided with a defective Ansible script. Your task is to find and fix bugs and provide the modified code with a reference for notification.* ***Defective Ansible script:" " "{defective_script}" " "***
Prompt2	*You will be provided with a defective Ansible script and bug report title. Your task is to find and fix bugs and provide the modified code with a reference for notification.* ***Issue report Title:" " "{issue_title}" " "*** ***Defective Ansible script:" " "{defective_script}" " "***
Prompt3	*You will be provided with a defective Ansible script and bug report title and body. Your task is to find and fix bugs and provide the modified code with a reference for notification.* ***Issue report Title:" " "{issue_title}" " "*** ***Issue report body:" " "{issue_body}" " "*** ***Defective Ansible script:" " "{defective_script}" " "***

Table 3 Characteristics of various LLMs

Type	ChatGPT 3.5	ChatGPT 4.0	Bard
Manufacturer	OpenAI	OpenAI	Google
Basic Model	GPT3.5	GPT4.0	PalM2
Parameters	1750B	Private	Private
Dataset (Word)	1.37T	1.37T	1.56T

Recently, LLMs developed independently by large companies have been actively announced. Except for Llama[8] provided by Meta, these announced LLMs are working in black-box natural, which means information on their architecture, layers, and parameters are private for potential commercial reasons. Furthermore, their enormous size makes them impractical for operation within local computing environments. Instead, they are designed to be operated via the web or the API provided by the manufacturer. Among various LLMs, we select three highly popular models, the characteristics of which are summarized in Table 3. OpenAI's ChatGPT 3.5 is the first LLM freely available on the web, and ChatGPT 4.0 is the most advanced system with more sophisticated language understanding and processing abilities than the previous version. On the other hand, Bard is a model announced to compete

[8]https://github.com/facebookresearch/llama

directly with ChatGPT and is also currently available for free use on the web. While OpenAI offers an API for accessing their LLMs, there is no officially provided API for Bard. Therefore, the three models experimented on the web and synthesized results.

4.3 Evaluation and Statistical Test

The evaluation process involved three Ph.D. students, each independently assessing a set of 63 cases, the number of cases initially collected. These assessments were conducted on three LLMs using three different prompts. They utilized objective and subjective indicators to evaluate LLMs' performance. The objective indicators are used to measure how closely LLM modifies the script to the human-modified script (i.e., Post-modified script), and there are four choices: (1) Does the script modified by the LLM according to the given prompt exactly match the human-modified code? If not, (2) Is it modifying the content inside the same task? (3) Amend the same line within the same task? (4) Amend irrelevant parts? The subjective indicator is used to assess how helpful the LLM-modified script is in fixing the actual defective script, and there are two choices: (1) Is the LLM-provided script helpful in modifying the actual defective script? or (2) isn't it?

Following the independent evaluation, we employed Krippendorff's α [11] to measure the agreement between evaluation results. It is a reliability coefficient that measures the agreement among observers or raters. Its values fall within the range of -1 to 1, with 1 representing perfect agreement between the raters, 0 indicating they are guessing randomly, and negative values suggesting the raters systematically disagree. The initial agreement rate of individual results was 0.58. However, all raters discussed cases where their evaluations diverged until Krippendorff's α was 1.0. Finally, of the 63 cases initially collected, 58 cases were used for analysis, excluding 5 cases in which the evaluations did not match.

5 Experimental Results

5.1 RQ1: Does Prompt Affect LLM Based ARP's Performance?

Table 4 illustrates the performance of three LLMs in ARP on various prompts. As described in Section 4.3, the performance analysis consists of objective and subjective evaluation. In the objective evaluation, we assess how closely the LLM responds to the human-modified code. "*Totally match*" indicates the case where the LLM's response is identical to the human-modified code.

Table 4 Performance analysis on various prompts over the three LLMs

# of cases (# of improved cases compare to prior prompt)

	Objective Evaluation				Subjective Evaluation	
	Totally Match	Task Match	Line Match	Miss	Helpful	Not Helpful
Prompt1	1 (−)	5 (−)	0 (−)	52 (−)	1 (−)	57 (−)
Prompt2	**3** (↑2)	**17** (↑12)	4 (↑4)	34 (↑18)	8 (↑7)	50 (↑7)
Prompt3	2 (↓1)	12 (↓5)	**12** (↑8)	**32** (↑2)	**15** (↑7)	**43** (↑7)

(a) Bard

# of cases (# of improved cases compare to prior prompt)

	Objective Evaluation				Subjective Evaluation	
	Totally Match	Task Match	Line Match	Miss	Helpful	Not Helpful
Prompt1	3 (−)	3 (−)	2 (−)	50 (−)	7 (−)	51 (−)
Prompt2	6 (↑3)	9 (↑6)	4 (↑2)	38 (↑12)	18 (↑11)	40 (↑9)
Prompt3	**7** (↑1)	**16** (↑7)	**10** (↑6)	**24** (↑14)	**26** (↑8)	**32** (↑8)

(b) ChatGPT v3.5

# of cases (# of improved cases compare to prior prompt)

	Objective Evaluation				Subjective Evaluation	
	Totally Match	Task Match	Line Match	Miss	Helpful	Not Helpful
Prompt1	1 (−)	0 (−)	4 (−)	53 (−)	4 (−)	54 (−)
Prompt2	3 (↑2)	9 (↑9)	15 (↑11)	31 (↑22)	20 (↑16)	38 (↑16)
Prompt3	**7** (↑4)	**10** (↑1)	**20**(↑5)	**22** (↑9)	**28** (↑8)	**30** (↑8)

(c) ChatGPT v4.0

"*Task match*" and "*Line match*" indicate the cases where the LLM's response is not the same as human-modified code, but it modified the same part of the human-modified code (i.e., in a task or line). Lastly, "*miss*" means LLM's response modifies parts unrelated to human-modified code. On the other hand, the subjective evaluation focuses on whether the LLM's response proves helpful in addressing the issue.

The first finding is that incorporating more information into the prompt generates more satisfactory code across all three models. This observation holds for objective and subjective evaluation indicators, with improvements evident as the amount of information in the prompt increases (i.e., prompt1 $\rightarrow$ prompt2 $\rightarrow$ prompt3). Even the addition of implicit information about the issue, as seen in prompt2 where only the issue title is included, results in significant enhancements in objective and subjective evaluation indicators compared to prompt1. Furthermore, when a detailed description of the issue (i.e., issue body) is added in prompt2, there is an increase in both indicators. However, this improvement is relatively less significant than

```
--- a/tasks/install.yml
+++ b/tasks/install.yml
@@ -2,7 +2,7 @@
- name: enable overcommit in sysctl
  sysctl:
    name: vm.overcommit_memory
-   value: 1
+   value: "1"
    state: present
    reload: yes
    ignoreerrors: yes
```

Figure 5 *git diff* result of case #13.

between prompt1 and prompt2. These results show that specifying the direction of automatic repairing LLM with additional information is important. Exceptionally, we can identify that Bard's objective evaluation performance degrades between prompt2 and prompt3. This degradation is attributed to Bard's token limitation, which restricts the amount of the given prompt. In the case of prompt3, there were cases where the entire Ansible script could not be entered in Bard, since the issue report body located in the middle of the prompt was too long, so it exceeded the token limitation. It prevents Bard from understanding the information of the entire Ansible script, resulting in poor performance compared to propmt2. In conclusion, the quantity of information included in the prompt provided to LLMs significantly influences their performance in ARP tasks.

The second finding highlights a limitation of LLMs ability to predict the presence of defects in a given Ansible script when provided with only the script information (i.e., prompt1), with some exceptions. Figure 5 shows the *git diff* result of case # 13, where the three LLM models modified the defective Ansible script to match the human-modified code precisely. In this case, the issue stemmed from a discrepancy in data types. Ansible script basically uses the *String* type as a variable. However, *1* (i.e., int type) is entered in the defective script, so the developer changed *1* as *Int* type into "*1*" as *String* type to resolve the issue. However, it is important to note that most cases involved issues not related to syntax errors in the script but specific conditions during script execution. Thus, it is essential to incorporate information about the specific conditions causing the issue into the prompt to address these cases (i.e., through prompt2 or prompt3). In conclusion, LLMs have limitations in predicting whether a given Ansible script contains defects when provided with script information alone. It is also imperative to include issue reports that provide context about the specific conditions leading to the issues.

The third finding is that it is difficult to directly apply LLM-based APR on Ansible script because only 28% (7 cases) are modified to be the same as the human-modified code, even when prompt3 is applied. In addition, in the case of subjective evaluation, the code generated by both versions of ChatGPT only helps with solving bugs in about 44% (26 cases) and 48% (28 cases) of all cases. However, the above findings indicate the potential for performance enhancement based on the format of the prompt and the information included. Consequently, there are plans for further research to improve performance by augmenting the prompt with additional information, such as utilizing few-shot learning, which involves providing examples of fixing similar issues.

In conclusion, we confirm that the performance of LLM varies depending on the given prompt, with more information provided in the prompt yielding better performance. However, further research is needed to enhance the quality of LLM's response. Furthermore, in subsequent analysis, we apply the subjective evaluation, which shows the difference in performance more clearly than the objective evaluation, and prompt3, which consistently shows the best performance among other prompts.

5.2 RQ2: Does Different LLM Show Similar Performance?

Table 5 summarizes the performance of each model, focusing on subjective indicators when using prompt3. As shown in the table, Bard's performance is lower than the other two versions of ChatGPT, with no significant disparity between the performance of the two ChatGPT versions. These results indicate that ChatGPTs comprehend Ansible scripts better than Bard and offer helpful responses in solving issues. However, it is important to note that model configuration and training data for all three models are private; analysis of the above performance differences cannot be performed. Nonetheless, considering that Ansible has a relatively simple structure compared to the other PL, it is reasonable to speculate that the amount of Ansible script in each training data may influence the models' performance.

Figure 8 shows a Venn diagram of cases where each model provides helpful responses for solving an issue. First, the three models provide helpful

Table 5 LLM's performance on subjective indicator using prompt3

	Bard	ChatGPT v3.5	ChatGPT v4.0
Helpful/Not Helpful	15/43	26/32	**28**/30

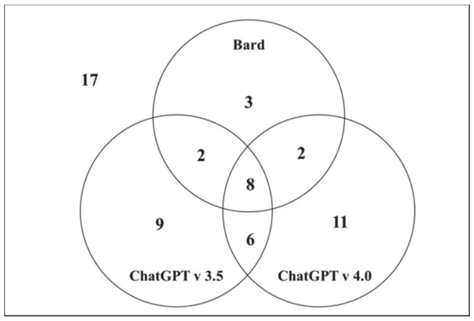


Figure 6 Venn diagram of cases where LLMs provide helpful responses.

Table 6 LLMs' performance analysis on various types of case

Helpful/Not helpful (Total case)

	Modification	Addition	Single Line	Multi Line	Single Task	Multi Task
Bard	6/24	9/19	5/ 14	10/29	9/26	6/17
	(30)	(28)	(19)	(39)	(35)	(23)
ChatGPTv3.5	12/18	**14**/14	6/13	18/21	14/21	**12**/11
	(30)	(28)	(19)	(39)	(35)	(23)
ChatGPTv4.0	**16**/14	12/16	**9**/10	**19**/20	**18**/17	10/13
	(30)	(28)	(19)	(39)	(35)	(23)

responses for issue resolution in 41 of the 58 cases (i.e., 70%). Though the problem of determining which LLM model's response is accurate remains, these results highlight the promise of LLM-based APR for Ansible scripts. Second, the models that provided helpful responses differ across the collected cases, even with the same prompt. There are cases where Bard, which shows relatively lower performance than the two versions of ChatGPT, provides more helpful responses than them, and similar patterns emerge with the two ChatGPT versions that show comparable performance between them. In addition, the cases in which the three models provided helpful responses in common were 13% (i.e., 8 cases) of the total. Based on these facts, providing prompts tailored to each model is essential because each model interprets a given prompt and responds differently. Finally, to check the differences between cases where responses are commonly helpful (i.e., 8 cases) and responses are not helpful (i.e., 17 cases), we analyze the characteristics of the given prompts, and the results will be discussed in Sector 5.

Table 6 shows the subjective evaluation results of the three models' performance across various types of cases when using prompt3. Similar to

Table 2, Bard does not outperform ChatGPT in any case. In addition, there are no notable differences between the two versions of ChatGPT depending on the type of case. However, ChatGPT v4.0 outperforms ChatGPT v 3.5, except in cases where code additions are required (i.e., *Addition*) or modified parts are distributed across multiple tasks (i.e., *Multi-Task*). Consequently, ChatGPT v4.0 outperforms other models, while there are no significant differences in the performance among models depending on the type of case.

In conclusion, among the three models, ChatGPT v4.0 is better than the others. Moreover, we can confirm that LLM-based ARP on Ansible is promising, although further investigation is necessary to determine the optimal model providing the most accurate responses.

6 Discussion

In this section, we analyze the relationship between the constituent elements of the prompt and the level of satisfaction of the responses provided by LLM. As explained in Section 3.2, Prompt3 consists of a defective Ansible script, an issue report title, and an issue report body. Given that the issue report title occupies a relatively minor portion of the prompt compared to the other components, we analyze the impact of the issue report body and defective Ansible script on the satisfaction levels associated with LLM responses.

Figure 7 is a dot graph showing cases where the three models provide helpful responses in common and cases where they do not (i.e., 8 cases and 17 cases in Section 4.3, respectively), according to the number of lines

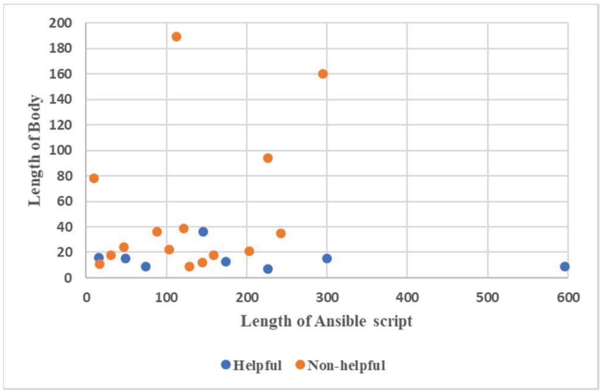
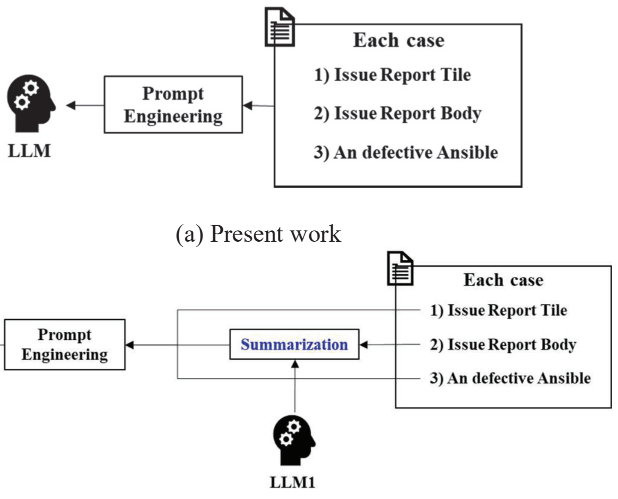


Figure 7 Correlation between the elements of the prompt and satisfaction of the response.

in the issue report body and the number of lines of the defective Ansible script. In the figure, regarding the number of lines of the defective Ansible script in the prompt, no discernible correlation is evident between it and the satisfaction level of the response. On the other hand, a significant correlation emerges between the length of the issue report body and the satisfaction level of the response. In cases where helpful responses were obtained (i.e., Orange dot in the figure), the number of lines in the issue report body is relatively small (i.e., 9∼36 lines). It indicates that the number of lines in the issue report, which LLM relies on to determine how to resolve the issue, has more influence than the line of Ansible script requiring correction. Additionally, if the issue report body in the middle of the prompt is too long, it exceeds the number of tokens LLM can remember while writing the response. Consequently, this leads to a decline in LLM performance because it loses the information necessary for reference during response generation.

In order to address this issue, we expect that satisfaction with LLM's response can be increased by providing a summary of the issue report body rather than providing it as is. As illustrated in Figure 8, we outline future work for an automated framework for Ansible. Figure 8(a) shows the currently applied framework, where an overly lengthy issue report body adversely affects response quality. On the other hand, Figure 8(b) shows our future



(a) Present work

(b) Future work

Figure 8 Frameworks of present and future work.

work's framework, which incorporates an additional LLM tasked with summarizing the issue report body. Through this approach, we aim to generate more effective prompts and enhance the performance of LLM-based APR for Ansible.

7 Threats to Validity

Internal Validity is the inconsistency of Bard and ChatGPTs' responses to the same question. The evaluation results could depend on repeated trials. To mitigate this threat, the three raters confirmed the results of the other raters through cross-validation, and we removed cases that did not match the evaluation results.

External Validity is that the small number of raters and cases (i.e., three raters and 58 cases, respectively, may limit the generalizability of our findings. To mitigate this threat, we applied Krippendorff's α to select only cases where the three evaluators were unanimous, and through this, we tried to overcome the small number of cases and raters. In future work, we plan to improve external Validity by involving more raters and cases.

8 Conclusion

Edge-Cloud system has a massively distributed infrastructure, and IaC is a crucial tool that helps deploy and manage the infrastructure of the edge-cloud system effectively. Ansible is one of the popular IaC tools; as Ansible is a set of code, its code quality influences the quality of the service delivered by the Edge-cloud system. We focused on LLM-based ARP to ensure the quality of the Ansible script. However, prior LLM-based APR studies have concentrated on widely used Programming Languages (PL), such as Java and C. Hence, this study evaluated the performance of LLM-based ARP on Ansible for the first time to confirm its applicability to Ansible. We assessed the performance of three LLMs with three types of prompts on 58 Ansible script revision cases. The results show that the LLMs provide d helpful responses in 70% of cases, which is promising, but further research is needed to apply it in practice.

In future work, we plan to apply few-shot learning in the prompt, which gives hints for issue resolution. Additionally, we plan to use an additional LLM that summarizes the overly extensive information in the prompt, which prevents LLMs from losing the information.

Acknowledgment

This research was supported by Information Technology Research Center (ITRC) support program supervised by the Institute of Information & Communications Technology Planning & Evaluation (IITP-2023-2020-0-01795), and Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2022R1I1A3069233).

References

[1] Agapito, G., Bernasconi, A., Cappiello, C., Khattak, H.A., Ko, I., Loseto, G., Mrissa, M., Nanni, L., Pinoli, P., Ragone, A., et al.: Current Trends in Web Engineering: ICWE 2022 International Workshops, BECS, SWEET and WALS, Bari, Italy, July 5–8, 2022, Revised Selected Papers. Springer Nature (2023)

[2] Ahmad, A., Waseem, M., Liang, P., Fahmideh, M., Aktar, M.S., Mikkonen, T.: Towards human-bot collaborative software architecting with chatgpt. In: Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering. pp. 279–285 (2023)

[3] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J.D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al.: Language models are few-shot learners. Advances in neural information processing systems 33, 1877–1901 (2020)

[4] Buyya, R., Srirama, S.N.: Fog and edge computing: principles and paradigms. John Wiley & Sons (2019)

[5] Dalla Palma, S., Di Nucci, D., Palomba, F., Tamburri, D.A.: Within-project defect prediction of infrastructure-as-code using product and process metrics. IEEE Transactions on Software Engineering 14(8), 1 (2020)

[6] Guerriero, M., Garriga, M., Tamburri, D.A., Palomba, F.: Adoption, support, and challenges of infrastructure-as-code: Insights from industry. In: 2019 IEEE International conference on software maintenance and evolution (ICSME). pp. 580–589. IEEE (2019)

[7] Hassan, M.M., Rahman, A.: As code testing: Characterizing test quality in open source ansible development. In: 2022 IEEE Conference on Software Testing, Verification and Validation (ICST). pp. 208–219. IEEE (2022)

[8] Jin, M., Shahriar, S., Tufano, M., Shi, X., Lu, S., Sundaresan, N., Svyatkovskiy, A.: Inferfix: End-to-end program repair with llms. arXiv preprint arXiv:2303.07263 (2023)
[9] Kabir, S., Udo-Imeh, D.N., Kou, B., Zhang, T.: Who answers it better? an in-depth analysis of chatgpt and stack overflow answers to software engineering questions. arXiv preprint arXiv:2308.02312 (2023)
[10] Kokuryo, S., Kondo, M., Mizuno, O.: An empirical study of utilization of imperative modules in ansible. In: 2020 IEEE 20Th international conference on software quality, reliability and security (QRS). pp. 442–449. IEEE (2020)
[11] Krippendorff, K.: Computing krippendorff's alpha-reliability (2011) 12. Kwon, S., Jang, J.I., Lee, S., Ryu, D., Baik, J.: Codebert based software defect prediction for edge-cloud systems. In: Current Trends in Web Engineering: ICWE 2022 International Workshops, BECS, SWEET and WALS, Bari, Italy, July 5–8, 2022, Revised Selected Papers. pp. 11–21. Springer (2023)
[12] Kwon, S., Jang, J.I., Lee, S., Ryu, D., Baik, J.: Codebert based software defect prediction for edge-cloud systems. In: Current Trends in Web Engineering: ICWE 2022 International Workshops, BECS, SWEET and WALS, Bari, Italy, July 5–8, 2022, Revised Selected Papers. pp. 11–21. Springer (2023)
[13] Kwon, S., Lee, S., Ryu, D., Baik, J.: Pre-trained model-based software defect prediction for edge-cloud systems. Journal of Web Engineering pp. 255–278 (2023)
[14] Kwon, S., Lee, S., Ryu, D., Baik, J.: Exploring the Feasibility of Chat-GPT for Improving the Quality of Ansible Scripts in Edge-Cloud Infrastructures through Code Recommendation: ICWE 2023 International Workshops, BECS, In proceeding
[15] Li, Z., Lu, S., Guo, D., Duan, N., Jannu, S., Jenks, G., Majumder, D., Green, J., Svyatkovskiy, A., Fu, S., et al.: Codereviewer: Pre-training for automating code review activities. arXiv preprint arXiv:2203.09095 (2022)
[16] Ma, W., Liu, S., Wang, W., Hu, Q., Liu, Y., Zhang, C., Nie, L., Liu, Y.: The scope of chatgpt in software engineering: A thorough investigation. arXiv preprint arXiv:2305.12138 (2023)
[17] Meijer, B., Hochstein, L., Moser, R.: Ansible: Up and Running. "O'Reilly Media, Inc." (2022)
[18] Monperrus, M.: Automatic software repair: A bibliography. ACM Computing Surveys (CSUR) 51(1), 1–24 (2018)

[19] Morris, K.: Infrastructure as code: managing servers in the cloud. "O'Reilly Media, Inc." (2016)
[20] Nascimento, N., Alencar, P., Cowan, D.: Comparing software developers with chatgpt: An empirical investigation. arXiv preprint arXiv:2305.11837 (2023)
[21] Nguyen, N., Nadi, S.: An empirical evaluation of github copilot's code suggestions. In: Proceedings of the 19th International Conference on Mining Software Repositories. pp. 1–5 (2022)
[22] Opdebeeck, R., Zerouali, A., De Roover, C.: Andromeda: A dataset of ansible galaxy roles and their evolution. In: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR). pp. 580–584. IEEE (2021)
[23] Opdebeeck, R., Zerouali, A., De Roover, C.: Smelly variables in ansible infrastructure code: detection, prevalence, and lifetime. In: Proceedings of the 19th International Conference on Mining Software Repositories. pp. 61–72 (2022)
[24] Pearce, H., Tan, B., Ahmad, B., Karri, R., Dolan-Gavitt, B.: Examining zero-shot vulnerability repair with large language models. In: 2023 IEEE Symposium on Security and Privacy (SP). pp. 2339–2356. IEEE (2023)
[25] Rahman, A., Rahman, M.R., Parnin, C., Williams, L.: Security smells in ansible and chef scripts: A replication study. ACM Transactions on Software Engineering and Methodology (TOSEM) 30(1), 1–31 (2021)
[26] Sobania, D., Briesch, M., Hanna, C., Petke, J.: An analysis of the automatic bug fixing performance of chatgpt. arXiv preprint arXiv:2301.08653 (2023)
[27] Tian, H., Lu, W., Li, T.O., Tang, X., Cheung, S.C., Klein, J., Bissyandé, T.F.: Is chatgpt the ultimate programming assistant–how far is it? arXiv preprint arXiv:2304.11938 (2023)
[28] White, J., Hays, S., Fu, Q., Spencer-Smith, J., Schmidt, D.C.: Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. arXiv preprint arXiv:2303.07839 (2023)
[29] Xia, C.S., Zhang, L.: Conversational automated program repair. arXiv preprint arXiv:2301.13246 (2023)
[30] Xia, C.S., Zhang, L.: Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. arXiv preprint arXiv:2304.00385 (2023)
[31] Xu, J., Yan, L., Wang, F., Ai, J.: A github-based data collection method for software defect prediction. In: 2019 6th International Conference on

Dependable Systems and Their Applications (DSA). pp. 100–108. IEEE (2020)

[32] Zhang, Y., Rahman, M., Wu, F., Rahman, A.: Quality assurance for infrastructure orchestrators: Emerging results from ansible. In: 2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C). pp. 1–3. IEEE (2023)

Biographies

Sunjae Kwon received the bachelor's degree in electric engineering from Korea Military Academy in 2009, the master's degree in computer engineering from Maharishi Markandeshwar University in 2015. He is a doctoral student in software engineering from KAIST. His research areas include software analytics based on AI, software defect prediction, mining software repositories, and software reliability engineering.

Sungu Lee received the bachelor's degree in mathematics from KAIST in 2021, the master's degree in software engineering from KAIST in 2022. He is a doctoral student in software engineering from KAIST. His research areas include software analytics based on AI, software defect prediction, mining software repositories, and software reliability engineering.

Taehyoun Kim received the bachelor's degree in information and computer engineering from Ajou university in 2012, the master's degree in computer science from KAIST in 2014. He is a full-time researcher at Agency for Defense Development and a doctoral student in computer science from KAIST. His research areas include software analytics based on AI, software defect prediction, mining software repositories, and software reliability engineering.

Duksan Ryu earned a bachelor's degree in computer science from Hanyang University in 1999 and a Master's dual degree in software engineering from KAIST and Carnegie Mellon University in 2012. He received his Ph.D. degree in school of computing from KAIST in 2016. His research areas include software analytics based on AI, software defect prediction, mining software repositories, and software reliability engineering. He is currently an associate professor in software engineering department at Jeonbuk National University.

Jongmoon Baik received his B.S. degree in computer science and statistics from Chosun University in 1993. He received his M.S. degree and Ph.D. degree in computer science from University of Southern California in 1996 and 2000 respectively. He worked as a principal research scientist at Software and Systems Engineering Research Laboratory, Motorola Labs, where he was responsible for leading many software quality improvement initiatives. His research activity and interest are focused on software six sigma, software reliability & safety, and software process improvement. Currently, he is a full professor in school of computing at Korea Advanced Institute of Science and Technology (KAIST). He is a member of the IEEE.