Code Smell-guided Prompting for LLM-based Defect Prediction in Ansible Scripts

Hyunsun Hong[1], Sungu Lee[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: jbaik@kaist.ac.kr
[*]*Corresponding Author*

Received 29 September 2024; Accepted 09 November 2024

Abstract

Ensuring the reliability of infrastructure as code (IaC) scripts, like those written in Ansible, is vital for maintaining the performance and security of edge-cloud systems. However, the scale and complexity of these scripts make exhaustive testing impractical. To address this, we propose a large language model (LLM)-based software defect prediction (SDP) approach that uses code-smell-guided prompting (CSP). In some cases, CSP enhances LLM performance in defect prediction by embedding specific code smell indicators directly into the prompts. We explore various prompting strategies, including zero-shot, one-shot, and chain of thought CSP (CoT-CSP), to evaluate how code smell information can improve defect detection. Unlike traditional prompting, CSP uniquely leverages code context to guide LLMs in identifying defect-prone code segments. Experimental results reveal that

Journal of Web Engineering, Vol. 23_8, 1107–1126.
doi: 10.13052/jwe1540-9589.2383

while zero-shot prompting achieves high baseline performance, CSP variants provide nuanced insights into the role of code smells in improving SDP. This study represents exploration of LLMs for defect prediction in Ansible scripts, offering a new perspective on enhancing software quality in edge-cloud deployments.

Keywords: Edge-cloud, Ansible, large language models, software defect prediction.

1 Introduction

As the edge-cloud ecosystem evolves, it has become a crucial platform for handling real-time, data-intensive applications in sectors like healthcare and autonomous systems. These applications demand minimal latency and robust infrastructure capable of efficiently processing large volumes of data [1,2]. At the heart of these systems lies infrastructure-as-code (IaC), a paradigm that enables automation of infrastructure management using machine-readable configuration files [3]. IaC facilitates seamless scaling and reproducibility, transforming how cloud-based services are provisioned and maintained. Ansible, one of the leading IaC tools, stands out for its simple YAML-based configuration and ease of use, making it a favored choice among developers.

In the context of edge-cloud environments, where the reliability and security of services are paramount, the accuracy of Ansible playbooks plays a critical role. Erroneous IaC scripts can lead to vulnerabilities, system outages, or inefficient resource utilization, further underlining the need for robust methods to ensure script accuracy. However, due to the complexity of edge-cloud systems and their large-scale nature, exhaustive testing of these scripts is neither practical nor cost-effective. This challenge has driven the need for software defect prediction (SDP) techniques that can efficiently identify defect-prone regions in code, enabling developers to focus their testing efforts on high-risk areas [4].

In our previous work [5], we introduced a methodology leveraging large language models (LLMs) to enhance SDP in Ansible scripts. Our approach incorporates code-smell-guided prompting (CSP), a technique that embeds known code smells into prompts provided to LLMs, improving their defect prediction accuracy. While the initial results were promising, the study was limited by the relatively small dataset and the outdated LLM models used. Furthermore, only two of the six commonly observed code smells in Ansible

projects, as identified by Opdebeeck et al. [6], were utilized as examples in the CSP-CoT (chain-of-thought) prompts.

This paper extends the previous study by addressing these limitations. First, we increased the size of the dataset, allowing for more generalized conclusions and a thorough evaluation of the proposed methodology. Second, we updated the experiments by employing the latest LLM architectures, ensuring that our analysis leverages current advancements in model performance and capabilities. Finally, we broadened the scope of code smells used in the CSP-CoT prompts, incorporating all six code smells defined by Opdebeeck et al. [6] to assess the impact of each smell type on defect prediction performance.

2 Background and Related Work

2.1 Code Smell in Ansible

Ansible is a widely used tool in the IaC domain due to its simplicity and YAML-based structure. However, its complex handling of variable precedence can lead to defective code, especially when not properly managed. A lack of comprehensive testing tools for Ansible further complicates ensuring correctness in infrastructure code. In their study, Opdebeeck et al. [6] identified six prevalent code smells in Ansible scripts. Among these code smells, the 'impure initializer' and 'unconditional override' are particularly problematic as they can inadvertently alter script execution. The impure initializer may unexpectedly modify a variable's value during initialization, while the unconditional override can replace pre-existing variable definitions without adequate validation

The connection between code smells and potential software defects is well-documented. Code smells often signal deeper structural issues within the codebase, leading to vulnerabilities or other failures [7]. While some may dismiss code smells as minor stylistic issues, their presence can have far-reaching implications for the stability and maintainability of code [8]. For instance, an improperly initialized variable in Ansible could trigger unexpected behaviors during deployment, potentially introducing defects that disrupt services. Similarly, an unconditional override might result in overwritten critical configurations, bypassing essential safety checks or altering the infrastructure's security posture. Understanding these code smells and addressing them early is crucial for developing robust, error-free Ansible scripts that maintain high code quality and system reliability standards.

2.2 Prompting Techniques for LLMs

The performance of LLMs is often influenced by the structure and quality of the prompts they are provided with. Prompt engineering, a critical area of research, has yielded various strategies to optimize LLM output. While simpler prompts have been advocated in some contexts, Brown et al. [9] demonstrated that a few-shot learning approach, where the prompt includes examples similar to the task at hand, can significantly improve model performance. More recently, the chain-of-thought (CoT) prompting technique [10] has gained attention for its ability to enhance the reasoning capabilities of LLMs by embedding a step-by-step logical approach within the prompt itself.

CoT has proven especially effective in tasks such as vulnerability detection and code analysis [11, 12], where reasoning about the process is crucial. Inspired by this, we propose the code-smell-guided prompting (CSP) technique, which incorporates code smell information into the prompts provided to LLMs. By embedding known code smells, CSP enables LLMs to understand better the context of SDP tasks, particularly in Ansible scripts. While previous studies, such as that of Kwon et al. [13], explored LLM-based program repair in Ansible, SDP remains an underexplored area within this domain.

Jin et al. [14] provided insights into improving LLM performance in program repair tasks by integrating the Infer static analyzer with a retrieval-augmented generation (RAG) framework [15], which augmented the prompts with bug reports. Their approach achieved impressive results in automatic program repair (APR) tasks. Similarly, this research hypothesizes that augmenting LLM prompts with results from Opdebeeck et al.'s [6] Ansible code smell static analyzer could yield improved results in SDP for Ansible scripts. This approach aims to bridge the gap in the existing literature by specifically targeting LLM-based SDP for Ansible.

3 Methodology

3.1 Building Dataset

3.1.1 Building test dataset using GitHub pull-request (GHPR)

In order to construct a dataset that reflects real-world software development practices, we utilized a GitHub pull-request (GHPR)-based system to gather instances of code changes. GHPR is a well-established workflow in open-source software development, where contributors propose code changes that undergo review and discussion before being merged into the main codebase.

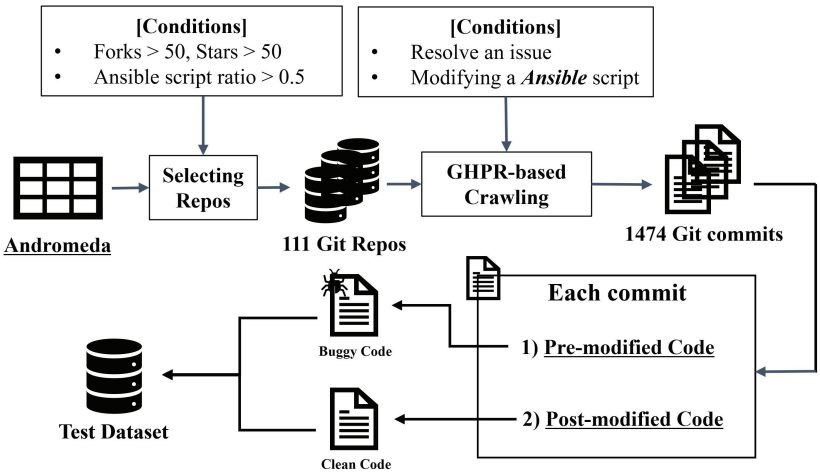

Figure 1 Overview of collecting the test dataset.

This workflow is particularly useful for identifying practical, issue-driven code modifications, making it ideal for building a dataset aimed at SDP. The starting point for our data collection was the Andromeda dataset [16], which consists of metadata from over 25,000 Ansible projects hosted on Ansible Galaxy[1], a widely-used platform for sharing Ansible roles and playbooks. From this extensive dataset, we applied specific selection criteria to filter out repositories that were relevant and active in the context of Ansible-based IaC.

The repository selection criteria included: (1) A repository with more than 50 forks and 50 stars to ensure the project is widely used and maintained. (2) Ansible scripts constituting more than 50% of the total codebase, ensuring that the repository is primarily focused on Ansible automation tasks. Compared to previous studies by Kwon et al. [13] and Hong et al. [5], which applied stricter criteria (such as requiring more than two core contributors, license requirements, and high issue frequency), our selection process was simplified to focus on repositories that met these two core conditions. This adjustment allowed for the inclusion of a broader range of projects while still ensuring that the repositories were of sufficient quality and relevance for our study. After applying these criteria, we identified 111 repositories that fit the selection parameters. Using a custom crawling tool designed for this task, we extracted the commit history of these repositories, specifically focusing on commits where contributors modified Ansible scripts to resolve issues. This

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

resulted in a dataset of 1474 commits, each corresponding to a bug fix or issue resolution through changes in Ansible scripts.

For each commit, the pre-modified code (the state of the code before the issue was resolved) was labeled as buggy, while the post-modified code (the state after the issue was resolved) was labeled as clean. The data-collection process is visualized in Figure 1, illustrating the flow from repository selection to the extraction of commit history. This study builds on previous work by Hong et al. [5], which used a smaller dataset of 39 commits for testing LLM-based defect prediction models. In contrast, our expanded dataset increases the number of test cases, allowing for a broader evaluation of LLM performance.

3.1.2 Building a dataset for demonstration learning

To evaluate the effectiveness of the CSP technique, we focused on the impact of different Ansible code smell types on LLM performance in defect prediction tasks. We conducted separate experiments for each of the six types of code smells identified by Opdebeeck et al. [6], which include: impure initializer, changed data dependence, unconditional override, unused override, unnecessary `set_fact`, and unnecessary `include_vars`. Piotrowski et al. [7] noted that not all types of code smell information are beneficial for defect prediction, as only specific types contribute to better predictive performance. This insight underscores the variability in how code smell types impact defect prediction outcomes. In the previous study by Hong et al. [5], only the two code smells deemed critical by the authors – namely, 'impure initializer' and 'unconditional override' – were utilized in the experiments. However, recognizing the potential for different code smell types to affect performance in varied ways, this study expands the investigation to include all six code smell types. By doing so, we aim to explore which specific types of code smell are most useful for improving defect prediction accuracy. Unlike the previous study by Hong et al. [5], which randomly selected examples from GHPR-extracted commits for CSP-CoT (chain-of-thought) prompting, this extended study isolates examples based on specific code smell types. Each code smell type was tested separately to observe its unique impact on the performance of LLM-based defect prediction.

To create this refined dataset, we used the validation data provided by Opdebeeck et al. [6] for their code smell checker, which includes 20 code examples per smell type, resulting in a total of 120 examples. These examples were used as part of the demonstration learning setup, serving as one-shot examples in one-shot and CSP-CoT prompting.

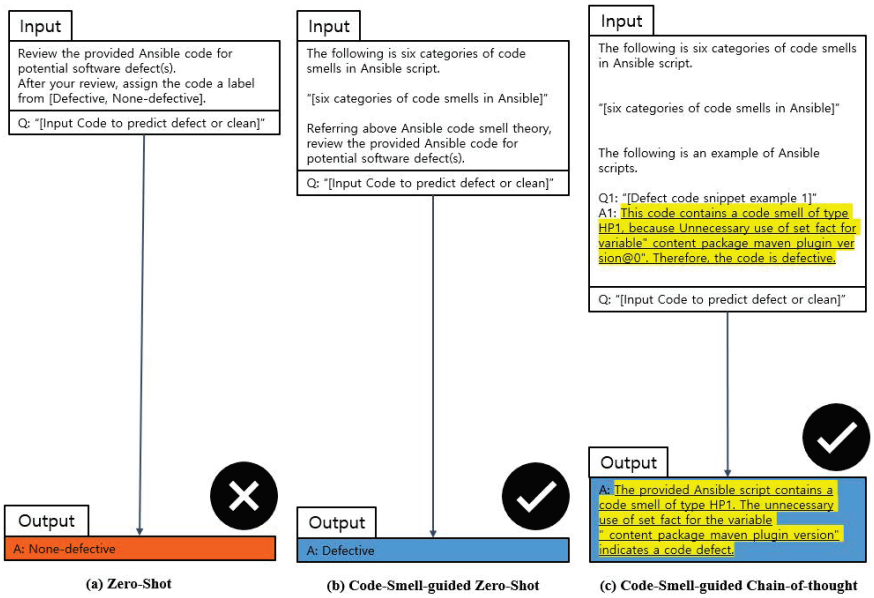

Figure 2 Examples of how CSP empowers LLMs to assess Ansible code by leveraging indicators of code smell. The CSP procedure is highlighted. Assume that ground truth of input code is buggy.

3.2 Code-smell-guided Prompting

The CSP technique aims to improve LLMs in predicting software defects in Ansible scripts by leveraging known code smell patterns. CSP incorporates detailed code smell information into the LLM prompt to enhance the model's understanding of the code structure and potential defects.

3.2.1 Overview of CSP

Figure 2 illustrates the overall flow of CSP, where the key components and steps of the methodology are outlined. CSP involves the identification of specific code smells in Ansible scripts, which are then integrated into the LLM prompts. This approach aims to improve the model's performance by providing it with additional contextual information based on the presence of code smells.

3.2.2 Procedure of CSP

The CSP methodology follows a structured process for incorporating code smell data into LLM prompts. The following steps outline this process:

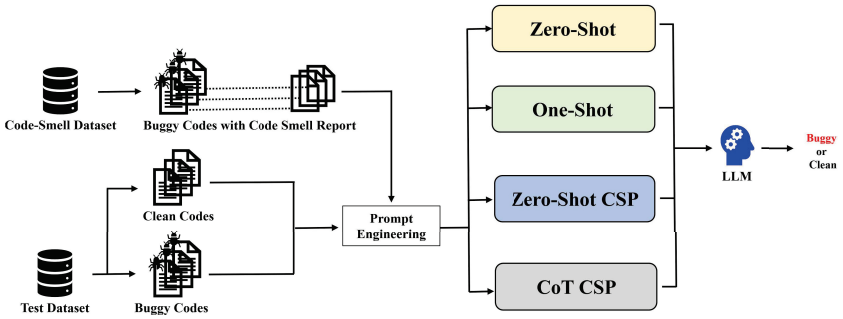

Figure 3 Evaluating CSP against traditional well-known strategies.

1. **Code smell identification**: An analysis is conducted on Ansible scripts to detect and classify prevalent code smells, such as 'impure initializer' and 'unconditional override,' as outlined by Opdebeeck et al. [6].
2. **Code smell information insertion**: Identified code smells are then encoded into a structured format suitable for inclusion in the LLM prompts. This insertion allows the LLM to interpret and utilize the code smell information effectively during prediction tasks.
3. **Prompt formulation**: Prompts are created that encompass both the original task (defect prediction) and the inserted code smell information, ensuring that the LLM is provided with comprehensive contextual data to facilitate more accurate predictions.

3.3 Evaluation of Prompting Strategies

Figure 3 illustrates the evaluation framework for CSP in comparison with well-known prompting strategies. Section 3.1 describes the test dataset, consisting of 1474 pairs of clean and buggy code from GitHub pull requests (GHPR), totaling 2948 test data points for evaluating CSP. To assess the CSP technique, we employed various prompting strategies: zero-shot, one-shot, zero-shot CSP, and chain-of-thought (CoT) CSP, each designed to evaluate LLM performance in defect prediction for Ansible code.

Zero-shot prompting predicts code defectiveness without examples, acting as a baseline for model generalization. One-shot prompting uses a labeled buggy code example to guide prediction, providing context for improved accuracy. Zero-shot CSP integrates code smell information from Opdebeeck et al. [6], offering an overview of Ansible code smells in the prompt to aid defect recognition. CoT CSP further enriches prompts by

combining buggy code examples with reasoning from code smell analyses, enhancing the model's step-by-step reasoning. The one-shot approach was chosen as a baseline to measure the impact of CSP on various code smell types. A shot example in one-shot prompting was randomly selected from Opdebeeck et al.'s validation dataset, regardless of code smell type. For the six CSP-CoT experiments, trials were conducted for each of the six code smells identified by Opdebeeck et al. [6], including impure initializer, changed data dependence, unconditional override, unused override, unnecessary `set_fact`, and unnecessary `include_vars`. Table 1 summarizes the prompting strategies used, with templates for zero-shot, one-shot, CSP zero-shot, and CSP CoT prompting. In Table 1, test data code is represented as 'input_code_snippet', with buggy example from the code-smell dataset inserted as 'example_buggy_code_snippet_1'. Code smells are categorized under 'warning_name', with their rationale and location detailed in 'warning_header'.

Unlike Hong et al. [5], which paired clean and defective code in few-shot prompting, we used one-shot prompting because the latest versions of code in GitHub repositories showed a lack of meaningful updates. Specifically, the 'clean' examples identified by Hong et al. were essentially unchanged from the code smell-checked versions validated by Opdebeeck et al. [6], implying that they might not actually represent clean code. This lack of updates suggests that these examples could be misleading as they may still contain underlying issues. To ensure accuracy, our approach focuses on a single correct example – a defective code snippet containing a code smell – rather than potentially unreliable 'clean' examples.

4 Experimental Setup

4.1 Research Questions

To assess the effectiveness of various prompting strategies in LLM-based SDP for Ansible scripts, we designed the following research questions (RQs).

RQ1: How do different prompting strategies affect the performance of LLMs in Ansible defect prediction, and what are the implications of their interactions?

RQ2: Among the different code smell-based CoT CSP strategies, which code smell type most significantly impacts defect prediction performance, and what insights can be derived from this?

Table 1 Prompt strategy variants

Type	Prompt template
Zero-shot	Review the provided Ansible code for potential software defect(s). **Code:** {`input_code_snippet`} **Label:**
One-shot	Review the provided Ansible code for potential software defect(s) based on the given example. **Code:** {`example_buggy_code_snippet_1`} **Label:** Defective **Code:** {`input_code_snippet`} **Label:**
Zero-shot CSP	The following are six categories of code smells in Ansible scripts. You can safely assume that the script is defective if it contains a code smell. 1. UR1: Due to an impure initializer, the variable value may have changed. ... 6. HP2: "include_vars" is unnecessarily used when it will be executed unconditionally. UR1 and UR2: Unsafe reuse UO1 and UO2: Unintended override HP1 and HP2: Too high precedence Referring to the above Ansible code smell theory, review the provided Ansible code for potential software defect(s). **Code:** {`input_code_snippet`} **Label:**
CoT CSP	The following are six categories of code smells in Ansible scripts. You can safely assume that the script is defective if it contains a code smell. 1. UR1: Due to an impure initializer, the variable value may have changed. ... 6. HP2: "include_vars" is unnecessarily used when it will be executed unconditionally. UR1 and UR2: Unsafe reuse UO1 and UO2: Unintended override HP1 and HP2: Too high precedence The following is an example of Ansible script. **Q1:** {`example_buggy_code_snippet_1`} **A1:** This code contains a code smell of type {`warning_name`}, because {`warning_header`}. Therefore, the code is defective. Let's think step by step. **Code:** {`input_code_snippet`} **Label:**

4.2 Dataset

In our experiment setup, we constructed a demonstration learning dataset with code examples labeled by code smell types as identified by Opdebeeck et al. [6]. Table 2 details the filtering process for the code smell prompting shot example dataset. Initially, 20 examples were collected for each of the six code smell types, totaling 120 examples. These underwent filtering steps: examples were excluded if their repository or specific commit was not found ('No repo' and 'No commit' rows). Token length was also verified, and any example exceeding the 128k token limit of the OpenAI GPT-4o Mini model was removed ('Token limit' row). After applying these filters, the final dataset retained 91 examples.

For token compatibility with LLMs, a similar filtering was applied to both the GHPR-derived test dataset. The GHPR-derived dataset's 1474 commits, each with paired clean and buggy code (2948 total snippets), were all within the 128k token limit. Prompts were generated for zero-shot, one-shot, zero-shot CSP, and CoT CSP strategies, covering all six code smell types. For one-shot and CoT CSP, an example from the demonstration learning dataset was randomly selected. To ensure adherence to the 128k token limit, if the incorporated prompt exceeded this threshold its example was replaced by another randomly chosen example from demonstration dataset until all prompts met the token constraint. This process was consistently applied across all code smell types and prompting strategies. Consequently, a total of 2948 test prompts were created across nine strategies. The dataset and code are publicly accessible for replication[2].

4.3 Large Language Models

This study evaluates three prominent LLMs, GPT-4o Mini, Gemini Flash, and Claude 3 Haiku, for their competitive performance on benchmarks and ability to handle large token limits. Table 3 compares their characteristics.

Table 2 Filtering process of a code smell prompting shot example dataset

Category	HP1	HP2	UO1	UO2	UR1	UR2
Original	20	20	20	20	20	20
No repo	2	1	0	0	0	1
No commit	2	3	6	3	5	1
Token limit	5	0	0	0	0	0
Final count	11	16	14	17	15	18

[2]https://github.com/HyunsunHong/CSP_for_LLM-based-SDP-in-Ansible

Table 3 Comparison of various LLMs

Type	GPT-4o Mini	Gemini Flash	Claude 3 Haiku
Manufacturer	OpenAI	Google	Anthropic
Input token limit	128K	1M	200K
Release date	18 July 2024	13 March 2024	23 July 2024
MMLU	82.0	78.9	76.7

The MMLU (massive multitask language understanding) benchmark measures a model's ability across diverse academic tasks. Scores for GPT-4o Mini, Gemini Flash, and Claude 3 Haiku are 82.0, 78.9, and 76.7, respectively. A crucial factor for choosing these LLMs is their large input token limits, with GPT-4o Mini at 128k, Gemini Flash at 1M and Claude 3 Haiku at 200k. These capacities exceed those used in previous work [5], and as more recent versions than those employed by Hong et al., these models show improved performance and increased input capacity. The ability to handle longer input sequences allows these LLMs to process a wider variety of test data.

5 Results

5.1 Data Filtering and Availability for LLM Responses

Table 4 outlines the data filtering process undertaken during the experiments for each combination of prompting strategy and LLM model. Initially, 2948 data points were prepared for every experiment involving a specific LLM and prompting strategy combination. Ideally, all prepared data points should have generated valid outputs for analysis. However, some exceptions occurred, resulting in two main categories of data reduction:

1. **Failed responses**: In some cases, Gemini failed to respond to certain input prompts due to internal safety mechanisms that blocked content generation for those queries. Such instances are labeled as 'Failed' in Table 4. The failure to receive a response meant that these data points were excluded from further analysis, as no outputs could be evaluated.
2. **Manual check required**: During the experiments, each input prompt was appended with a guiding instruction to conclude the response with a label like '[defect]' or '[clean].' This was done to facilitate automatic evaluation of the outputs. If the generated output lacked these labels, it required manual checking to determine the correct inference. However,

Table 4 Data filtering process for LLM responses and manual checks

	GPT-4o Mini				Gemini Flash				Claude 3 Haiku			
Prompting strategy	Prepared	Failed	Manual check	Total	Prepared	Failed	Manual Check	Total	Prepared	Failed	Manual Check	Total
Zero-shot	2948	0	7	2941	2948	429	451	2068	2948	0	995	1953
One-shot	2948	0	134	2814	2948	292	418	2238	2948	0	1439	1509
Zero-shot CSP	2948	0	32	2916	2948	0	313	2635	2948	0	155	2793
CoT CSP UR1	2948	0	0	2948	2948	0	233	2715	2948	0	0	2948
CoT CSP UR2	2948	0	2	2946	2948	0	177	2771	2948	0	15	2933
CoT CSP UO1	2948	0	1	2947	2948	0	178	2770	2948	0	0	2948
CoT CSP UO2	2948	0	2	2946	2948	0	231	2717	2948	0	0	2948
CoT CSP HP1	2948	0	2	2946	2948	0	194	2754	2948	0	0	2948
CoT CSP HP2	2948	0	2	2946	2948	0	366	2582	2948	0	2	2946

due to limited resources for manual validation, these outputs were also excluded from the analysis. The 'Manual Check' column in Table 4 represents the number of outputs that fell into this category.

The 'Total' column for each LLM and strategy reflects the final number of data points available for analysis after accounting for failed responses and those requiring manual review.

This filtering process, primarily affected by Gemini's safety mechanism and the need for manual validation, resulted in different numbers of usable data points across prompting strategies and LLMs. The final available data ('Total' column) provides the basis for evaluating and comparing the performance of the prompting strategies in subsequent sections.

5.2 Overall Performance Across LLMs

Table 5 provides an overview of the performance across various prompting strategies – zero-shot, one-shot, zero-shot CSP, and CoT CSP – evaluated on three LLMs: GPT-4o Mini, Gemini Flash, and Claude 3 Haiku. The highest precision, recall, and F1 score for each LLM are highlighted in bold. Across all models, zero-shot prompting generally outperformed other strategies in terms of recall and F1 score, showing its effectiveness for LLM-based defect prediction without predefined guidance.

One-shot prompting, which introduces a single example, generally performed better than zero-shot in F1 scores for both Gemini Flash and Claude 3 Haiku, although it slightly lagged behind zero-shot in GPT-4o Mini. Zero-shot CSP, which incorporates code smell information directly into the prompt, showed results comparable to one-shot but did not present a significant improvement. Interestingly, CoT CSP strategies, which were expected to enhance model reasoning through detailed breakdowns of code smell types, exhibited reduced effectiveness, often falling below the performance of both zero-shot and one-shot strategies. This underperformance indicates that

Table 5 Performance of different prompting strategies across LLMs

	GPT-4o Mini			Gemini Flash			Claude 3 Haiku		
Prompting strategy	Precision	Recall	F1 score	Precision	Recall	F1 score	Precision	Recall	F1 Score
Zero-shot	0.50	**0.87**	**0.63**	0.49	0.89	0.64	**0.50**	0.59	0.54
One-shot	0.50	0.78	0.61	0.50	**0.95**	**0.65**	**0.50**	0.86	0.63
Zero-shot CSP	0.50	0.83	0.62	0.50	0.92	**0.65**	**0.50**	**0.94**	**0.65**
CoT CSP UR1	0.49	0.56	0.53	**0.51**	0.77	0.61	**0.50**	0.42	0.46
CoT CSP UR2	0.50	0.65	0.56	0.50	0.83	0.62	**0.50**	0.50	0.50
CoT CSP UO1	0.49	0.49	0.49	0.50	0.86	0.63	**0.50**	0.47	0.48
CoT CSP UO2	0.49	0.43	0.46	0.50	0.85	0.63	0.49	0.48	0.48
CoT CSP HP1	**0.51**	0.50	0.50	0.49	0.80	0.61	0.48	0.44	0.46
CoT CSP HP2	0.49	0.48	0.49	**0.51**	0.48	0.49	0.49	0.35	0.41

detailed prompts focusing on specific code smells may narrow the reasoning ability of LLMs, leading to less effective defect predictions.

5.3 Answering RQ1: Effectiveness of CSP vs. Standard Prompting

The results show that zero-shot prompting achieved the highest F1 score for GPT-4o Mini, while one-shot prompting led in Gemini Flash, closely followed by zero-shot CSP, and zero-shot CSP was most effective in Claude 3 Haiku. However, the performance differences between zero-shot and one-shot in GPT-4o Mini and Gemini Flash were trivial, whereas in Claude 3 Haiku, zero-shot performance was notably lower compared to one-shot and zero-shot CSP. This indicates that the best prompting strategy may vary by model, with some models benefiting more from guided context than others.

These findings suggest that unrestricted prediction in zero-shot prompting allows LLMs to flexibly assess code defects, leading to comprehensive identification in some contexts. In contrast, one-shot prompting can enhance performance for certain LLMs by providing guiding examples without overly constraining flexibility. Additionally, incorporating code smell-based rules, as in zero-shot CSP, can offer helpful context akin to examples in one-shot prompting.

However, when examples are combined with specific rules, as in CoT CSP strategies, performance often declines. This strategy's emphasis on step-by-step reasoning for particular code smells seems to narrow the LLMs' focus, hindering their ability to generalize and identify a wider range of defects effectively. Our refined analysis acknowledges that while CSP techniques can enhance performance in particular scenarios, their utility is not consistent across all LLMs and prompting strategies.

5.4 Answering RQ2: Impact of Different CoT CSP Code Smell Types

In examining RQ2, we analyzed the performance of CoT CSP strategies across the six code smell types (UR1, UR2, UO1, UO2, HP1, HP2). As shown in Table 5, the results indicate that not all code smells contribute equally to defect prediction. While the 'Changed data dependence' (UR2) code smell demonstrated relatively strong recall and F1 scores across models, particularly in GPT-4o Mini and Claude 3 Haiku, it did not always achieve the highest performance compared to other CoT CSP strategies, such as in Gemini Flash. However, UR2 did provide relatively high performance among CoT CSP types, suggesting that context around variable dependencies is generally helpful for defect identification.

This conclusion aligns with the observations by Piotrowski et al. [7], who emphasized that the utility of code smell information in defect prediction varies significantly by type. Consequently, future work should focus on identifying the most impactful code smells for LLM-based SDP and exploring optimal ways to integrate them into prompting strategies.

5.5 Implications and Future Directions

The results reveal that while zero-shot prompting often provides strong performance in SDP for Ansible scripts, it is not universally the best strategy. For instance, in Gemini Flash, one-shot prompting slightly outperformed zero-shot, and in Claude 3 Haiku, zero-shot CSP achieved the highest F1 score. This indicates that providing context through a single example or incorporating code smell information directly into the prompt can improve defect prediction performance in some models without overly constraining their reasoning capabilities. The varied results suggest that there is no one-size-fits-all prompting strategy, as different LLMs respond to prompt context differently. The relatively strong and consistent performance of zero-shot CSP across multiple LLMs underscores the potential value of incorporating code smell information into prompting. However, more structured CoT CSP strategies generally underperformed, possibly due to the narrowed focus on step-by-step reasoning, which might hinder the model's ability to generalize defect prediction across a broader code context.

These findings highlight the need to further explore and refine the balance between specificity and flexibility in prompting LLMs. Future research should investigate how best to combine various prompting techniques, such

as examples and rules, or adjust prompts dynamically based on code characteristics to optimize defect prediction in different contexts and LLM architectures.

6 Threats to Validity

Our study faced two primary limitations. First, the Gemini Flash model triggered safety mechanisms for certain prompts, blocking content generation and resulting in incomplete test coverage. This may have skewed performance evaluation. Second, only outputs with explicit labels ([clean], [cleaness], [defect], [defective]) were included in the results. Ambiguous outputs were excluded without manual verification due to time constraints, potentially affecting result accuracy. Addressing these issues in future work could improve the reliability of the findings.

7 Conclusion

This study explores the impact of various prompting strategies for LLMs on SDP in Ansible scripts, focusing on zero-shot, one-shot, and code smell-guided (CSP) prompts. The findings reveal that the optimal prompting strategy varies across LLMs: zero-shot prompting performed best for GPT-4o Mini, while one-shot prompting and zero-shot CSP were slightly superior in Gemini Flash, and zero-shot CSP led in Claude 3 Haiku. However, the performance differences were generally small, except in Claude 3, where CSP significantly improved defect detection. Moreover, while embedding code smell rules through zero-shot CSP was found to aid SDP, step-by-step CoT CSP prompting often reduced performance, likely due to its constrained focus. The evaluation of specific code smell types highlighted 'changed data dependence' (UR2) as a relatively effective guide, although it did not always outperform other strategies. This underscores the importance of balancing prompt context and model flexibility to enhance defect prediction. The work lays the foundation for refining prompt design, integrating code smells, and constructing GHPR-based Ansible SDP datasets to better leverage LLM capabilities for defect detection in infrastructure-as-code contexts like Ansible. Future research should seek a dynamic approach to prompt formulation, adjusting based on both LLM characteristics and code context for more accurate SDP.

Acknowledgement

This research was supported by the MSIT(Ministry of Science and ICT), Korea, under the ITRC(Information Technology Research Center) support program(IITP-2024-2020-0-01795) supervised by the IITP(Institute for Information & Communications Technology Planning & Evaluation), Basic Science Research Program through the National Research Founda-tion of Korea(NRF) funded by the Ministry of Education(NRF-2022R1I1A3069233), and the Korea Agency for Infrastructure Technology Advancement (KAIA) grant funded by the Ministry of Land, Infrastructure and Transport (Grant RS-2021-KA163348).

References

[1] Giuseppe Agapito, Anna Bernasconi, Cinzia Cappiello, Hasan Ali Khattak, InYoung Ko, Giuseppe Loseto, Michael Mrissa, Luca Nanni, Pietro Pinoli, Azzurra Ragone, 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] Kief Morris. *Infrastructure as code: managing servers in the cloud*. "O'Reilly Media, Inc.", 2016.

[3] Bas Meijer, Lorin Hochstein, and René Moser. *Ansible: Up and Running*. "O'Reilly Media, Inc.", 2022.

[4] Romi Satria Wahono. A systematic literature review of software defect prediction. *Journal of software engineering*, 1(1):1–16, 2015.

[5] Hyunsun Hong, Sungu Lee, Duksan Ryu, and Jongmoon Baik. Enhancing software defect prediction in ansible scripts using code-smell-guided prompting with large language models in edge-cloud infrastructures. In *Proceedings of the 4th International Workshop on Big data driven Edge Cloud Services (BECS 2024) Co-located with the 24th International Conference on Web Engineering (ICWE 2024)*, Tampere, Finland, June 17–20 2024.

[6] Ruben Opdebeeck, Ahmed Zerouali, and Coen De Roover. Smelly variables in ansible infrastructure code: Detection, prevalence, and lifetime. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 61–72, 2022.

[7] Paweł Piotrowski and Lech Madeyski. Software defect prediction using bad code smells: A systematic literature review. *Data-centric business*

and applications: towards software development (volume 4), pages 77–99, 2020.

[8] Phongphan Danphitsanuphan and Thanitta Suwantada. Code smell detecting tool and code smell-structure bug relationship. In *2012 Spring congress on engineering and technology*, pages 1–5. IEEE, 2012.

[9] Tom B Brown. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[10] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[11] Yu Nong, Mohammed Aldeen, Long Cheng, Hongxin Hu, Feng Chen, and Haipeng Cai. Chain-of-thought prompting of large language models for discovering and fixing software vulnerabilities. *arXiv preprint arXiv:2402.17230*, 2024.

[12] Rasmus Ingemann Tuffveson Jensen, Vali Tawosi, and Salwa Alamir. Software vulnerability and functionality assessment using llms. In *2024 IEEE/ACM International Workshop on Natural Language-Based Software Engineering (NLBSE)*, pages 25–28. IEEE, 2024.

[13] Sunjae Kwon, Sungu Lee, Taehyoun Kim, Duksan Ryu, and Jongmoon Baik. Exploring the feasibility of chatgpt for improving the quality of ansible scripts in edge-cloud infrastructures through code recommendation. In *International Conference on Web Engineering*, pages 75–83. Springer, 2023.

[14] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. Inferfix: End-to-end program repair with llms. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1646–1656, 2023.

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

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

Biographies

Hyunsun Hong received his bachelor's degree in computer science and electrical engineering from Handong Global University in 2023. He is a master's student in software engineering at KAIST. His research areas include software analytics based on AI and software defect prediction.

Sungu Lee received his bachelor's degree in mathematics from KAIST in 2021 and his 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.

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.Sc. degree in computer science and statistics from Chosun University in 1993. He received his M.Sc. 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 and 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.