A comparative study of large language models with chain-of-thought prompting for automated program repair

Eko Darwiyanto[1,3], Rizky Akbar Gusnaen[1], Rio Nurtantyana[1,2,4]
[1]Informatics Study Program, School of Computing, Telkom University, Bandung, Indonesia
[2]Research Center for Data and Information Sciences, National Research and Innovation Agency, Bandung, Indonesia
[3]Center of Excellence HUMIC, School of Computing, Telkom University, Bandung, Indonesia
[4]Center of Excellence of Artificial Intelligence for Learning and Optimization, Telkom University, Bandung, Indonesia

Article Info

Article history:

Received Jan 28, 2025
Revised Jul 11, 2025
Accepted Oct 19, 2025

Keywords:

Automated program repair
Chain-of-thought prompting
Large language models
Quixbugs
Standard prompting

ABSTRACT

Automatic code repair is an important task in software development to reduce bugs efficiently. This research focuses on developing and evaluating a chain-of-thought (CoT) prompting approach to improve the ability of large language models (LLMs) in automated program repair (APR) tasks. CoT prompting is a technique that guides LLM to generate step-by-step explanations before providing the final answer, so it is expected to improve the accuracy and quality of code repair. This research uses the QuixBugs dataset to evaluate the performance of several LLM models, including DeepSeek-V3 and GPT-4o, with two prompting methods, namely standard and CoT prompting. The evaluation is based on the average number of plausible patches generated as well as the estimated token usage cost. The results show that CoT prompting improves performance in most models compared with the standard. DeepSeek-V3 recorded the highest performance with an average of 36.6 plausible patches and the lowest cost of $0.006. GPT-4o also showed competitive results with an average of 35.8 plausible patches and a cost of $0.226. These results confirm that CoT prompting is an effective technique to improve LLM reasoning ability in APR tasks.

Corresponding Author:

Rio Nurtantyana
Research Center for Data and Information Sciences, National Research and Innovation Agency (BRIN)
Bandung, Indonesia
Email: akunerio@gmail.com

1. INTRODUCTION

Software development is an inherently complex process prone to errors. Bugs or defects in software can result in serious consequences, ranging from user inconvenience to significant financial losses [1]. Thus, bug fixing is a critical task in the software development lifecycle. However, manual bug fixing is often time-consuming and resource-intensive [2]. This is where automated program repair (APR) plays a crucial role in enhancing efficiency and reducing bug-fixing costs.

In recent years, APR has become an active area with various approaches proposed, including search-based, constraint-based, template-based, and learning-based methods [3], [4]. One of the latest advancements in APR involves the utilization of large language models (LLMs) such as GPT-3 and Codex [5]. Codex is a model developed by OpenAI and designed to handle various programming tasks, including code generation, problem-solving, and bug fixing [6]. Additionally, Codex serves as the backbone of GitHub Copilot, a tool that aims developers in writing code more quickly and efficiently [5]. Despite these advancements, the performance of LLMs in bug fixing can be further improved. Prenner *et al.* [7] evaluated Codex's bug-fixing capabilities using the QuixBugs dataset and found that prompts significantly impact

Codex's performance. Conversely, inappropriate prompts may cause Codex to produce incorrect fixes. Therefore, developing effective prompting techniques is essential to enhance LLM performance in APR.

A prompt is a set of instructions provided to an LLM to generate specific responses or outputs [8]. In the context of LLM, prompts are often designed with a technique known as prompt engineering. Prompt engineering involves systematically designing prompts to guide LLMs in generating desired outputs or achieving particular goals [8]. This technique involves crafting effective patterns or templates to help models understand the given task, especially in cases that require reasoning or complex problem-solving. However, the prompt engineering concept that has yet to be extensively explored in APR is chain-of-thought (CoT) prompting. CoT prompting is a technique that guides LLMs to produce step-by-step explanations before providing a final answer [9], [10]. This approach has proven effective in enhancing LLM performance in various tasks, such as arithmetic and problem-solving [11]. CoT prompting enables LLMs to decompose complex problems into simpler sub-problems and provide more structured explanations [12]. In the context of APR, CoT prompting can assist LLMs in better understanding the bug context, identifying its causes, and generating correct fixes.

Hence, this study aims to explore the potential and design effective CoT prompting techniques, based on relevant literature to improve LLM performance in APR. In this study, the proposed CoT prompt structure will be compared with the standard prompting approach to evaluate performance improvement and cost efficiency. Standard prompting is a straightforward approach where a model is given a question or problem and directly produces a final answer. Conversely, CoT prompting guides the model to provide a step-by-step explanation before generating the final answer. The evaluation will use the QuixBugs dataset [13], which is a benchmark designed to test APR capabilities. Furthermore, the QuixBugs dataset has been widely used in various APR studies to assess code repair techniques [14]. In this study, the evaluation results include an analysis of the LLM's performance in generating correct code repairs on the QuixBugs dataset, as well as a cost estimate based on the number of tokens used. Therefore, the contribution of this study is to develop the structure of CoT prompting for APR and to compare the performance of various LLM models using the standard prompting and our proposed CoT prompting.

2. METHOD

2.1. Dataset

The dataset employed in this study is QuixBugs, a benchmark comprising 40 programs with bugs implemented in both Python and Java [15]. QuixBugs was selected because it provides well-defined test cases that can be utilized to evaluate the outcomes of code repairs [14]. The bugs present in QuixBugs encompass a wide range of common errors in software development, such as logical, arithmetic, and function call errors.

2.2. Utilizing the LLMs models with API

This study utilizes various LLMs to assess the effectiveness of CoT prompting for APR tasks, as listed in Table 1. We used 10 public LLMs that were released from mid until the end of 2024 and it could be accessed via API, such as GPT-4o, o1-Preview, o1-mini, Claude-3.5-Sonnet, Llama-3.3-70B, Gemini-1.5-Pro, Gemini-1.5-Flash, Grok-Beta, and Grok-2. These models were selected based on the quality index published by artificial analysis [16]. Additionally, this study incorporates the DeepSeek-V3 model, a new model demonstrating significant potential in code repair tasks [17]. Hence, this study can evaluate the effectiveness of CoT prompting across models with different approaches.

Table 1. The LLM models used in this study

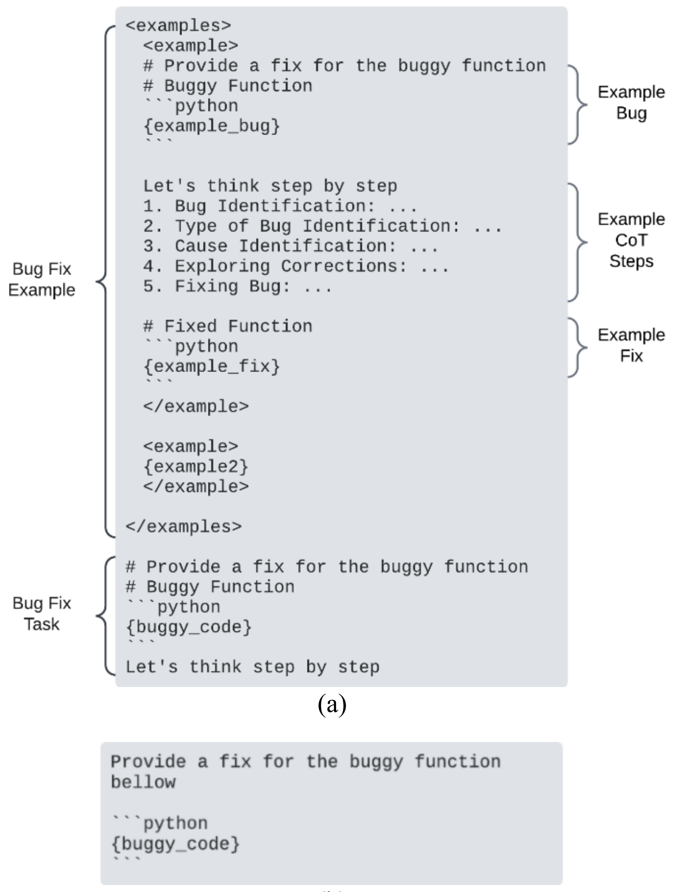
Model	API provider	Total params	Open source	Release date (2024)	Knowledge cutoff	Price/1M Token (USD)		API endpoint
						Input	Output	
GPT-4o	Azure	-	No	06 Aug.	Oct. 23	2.5	10	https://riakgu.openai.azure.com
Grok-Beta	xAI	-	No	13 Aug.	-	5	15	https://api.x.ai
Grok-2	xAI	-	No	13 Aug.	-	2	10	https://api.x.ai
o1-mini	OpenAI	-	No	12 Sep.	Oct. 23	3	12	https://api.openai.com
o1-Preview	OpenAI	-	No	12 Sep.	Oct. 23	15	60	https://api.openai.com
Gemini-1.5-Flash	Google	-	No	24 Sep.	Aug. 24	.075	.3	https://generativelanguage.googleapis.com
Gemini-1.5-Pro	Google	-	No	24 Sep.	Aug. 24	1.25	5	https://generativelanguage.googleapis.com
Claude-3-5-Sonnet	Anthropic	-	No	22 Oct.	Apr. 24	3	15	https://api.anthropic.com
Llama-3.3-70B	Azure	70 B	Yes	06 Dec.	-	.71	.71	https://riakgu2.services.ai.azure.com
DeepSeek-V3	Deepseek	670 B	Yes	27 Dec.	-	.014	.28	https://api.deepseek.com

Notes: The symbol "-" indicates unknown data; data has been updated and processed by the researcher as of January 16, 2025.

The parameter settings in this study adopt a Greedy sampling approach [18], where the temperature is set to 0.0 and top_p to 0.01. This approach ensures that the model consistently selects the token with the highest probability at each step, resulting in consistent output. Greedy sampling has been used by Lee *et al*. [19] and has proven to be superior in automatic scoring tasks using the CoT technique.

2.3. The structure of the proposed CoT prompt and standard prompt

The proposed of the CoT prompt in this study is a modification of the approach in [19], [20]; which implemented few-shot prompting for APR tasks. This research adapts that approach by converting it into CoT prompting designed to guide the model in resolving problems incrementally. Figure 1 shows the difference prompt structures. The CoT prompt used in this study comprises two main sections such as bug fix example and bug fix task, as illustrated in Figure 1(a). Meanwhile the standard prompt was used as standard comparison as illustrated in Figure 1(b).


Bug Fix Example
Example Bug
Example CoT Steps
Example Fix
Bug Fix Task

~~~
<examples>
  <example>
  # Provide a fix for the buggy function
  # Buggy Function
  ```python
  {example_bug}
  ~~~

  Let's think step by step
  1. Bug Identification: ...
  2. Type of Bug Identification: ...
  3. Cause Identification: ...
  4. Exploring Corrections: ...
  5. Fixing Bug: ...

  # Fixed Function
  ```python
  {example_fix}
  ```
  </example>

  <example>
  {example2}
  </example>

</examples>
Provide a fix for the buggy function
Buggy Function
```python
{buggy_code}
```
Let's think step by step
```

(a)

```
Provide a fix for the buggy function
bellow

```python
{buggy_code}
```
```

(b)

Figure 1. The difference prompt structures, such as (a) the proposed CoT prompt and (b) the standard prompt

The bug fix example section consists of two manually created code repair examples, each containing three primary elements such as example bug, example CoT steps, and example fix. Example bug includes a code snippet with a bug that requires correction. Example CoT steps begin with the phrase "Let's think step by step," as proposed by Kojima *et al*. [21], followed by five steps adopted from the debugging process examined by Asadollah *et al*. [22] such as i) bug identification, ii) type of bug identification, iii) cause identification, iv) exploring corrections, and v) fixing bug. In the last part, the example fix is the snippet of code that has been corrected following the CoT process. Furthermore, the entire bug fix example section is organized in a structured format using <examples> and <example> tags, in line with Anthropic's [23] recommendations to enhance clarity and organization in the prompt structure.

The bug fix task section constitutes the main portion of the prompt provided to the model. In this section, the model is tasked with applying the repair steps learned from the bug fix example to the code containing a bug. The model is expected to adhere to the CoT approach by systematically analyzing the bug, beginning with bug identification, determining the type of bug, identifying the cause, exploring solutions, and
```

implementing the fix. Consequently, the model not only produces corrected code but also provides a step-by-step explanation of the solution offered. This ensures that code corrections are carried out with a deep understanding and a structured analytical process, resulting in more accurate solutions. Conversely, the standard prompt used in this study, which adopts a general approach like zero-shot prompting [24]. In this approach, the model is assigned the task of repairing buggy code without additional guidance or prior examples.

**2.4. The system development of the automated program repair**

The APR system was developed using the Python programming language and the APR process is visualized through a flowchart, as in Figure 2, which outlines the workflow from dataset reading to generating token usage statistical summaries for cost estimation. The process begins by reading a dataset containing a collection of buggy files. If the dataset is empty, the system immediately terminates the process without proceeding with subsequent steps. However, if the dataset contains files, the system will read the first available buggy file. Once the buggy file is read, the system constructs a prompt using a pre-prepared template. The CoT or standard prompt template is submitted to the LLM through an API to generate a response consisting of the repaired code. The response is extracted to obtain the relevant code segment that serves as the solution for the identified bug. This repaired code is saved for validation in the subsequent stage. Additionally, the APR system records the number of tokens used by the model in generating the response to estimate the cost of LLM service usage based on the applicable token pricing according to the LLMs.

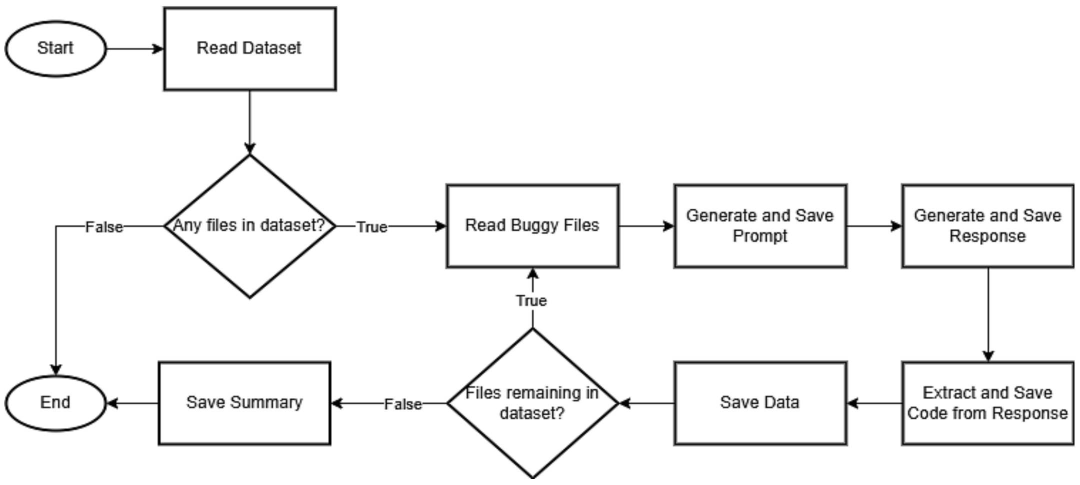


Figure 2. APR workflow

**2.5. The validation and evaluation methods**

The validation stage aims to evaluate the code produced by the APR process to ensure that the bugs have been correctly fixed. The workflow for the validation process is systematically depicted in the flowchart, as in Figure 3, starting from code reading to the storage of test results. Before validation begins, the code extracted from the LLM response is manually reviewed to ensure that only relevant sections are used. This step is crucial because LLM responses often lack structure or contain additional information that is not pertinent to the testing. Once the review is complete and the code is adjusted, validation proceeds by reading the code for testing. In detail, the validation process employs Pytest [25] to evaluate the code against a series of test cases provided in the QuixBugs dataset. Pytest generates a testing log that records details of each test case, including the number of test cases passed or failed. Upon completion of all tests, the Pytest log is processed through a parsing stage to produce a structured report.

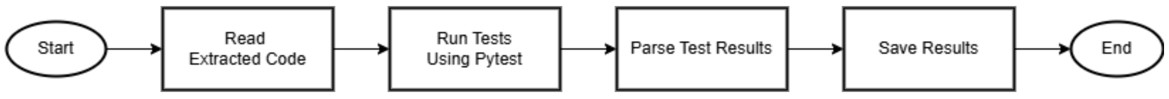


Figure 3. The validation workflow

In terms of evaluation, this study evaluates the outcomes of code repair using the metric of plausible patches [26]. A patch is deemed plausible if the resulting code correction passes all the test cases provided in the test suite [27]. By employing plausible patches as a metric, this research can assess the model's ability to produce code repairs that conform to the specifications without introducing new bugs.

## 3. RESULTS AND DISCUSSION

The results were conducted by comparing two prompting methods, such as standard prompting and proposed CoT prompting. Each model was tested five times on the QuixBugs dataset, which contains 40 buggy programs. The evaluation involved calculating the average number of plausible patches produced and the average token usage for each testing scenario.

### 3.1. The comparison of model performance based on the number of plausible patches

Model performances were measured by the number of plausible patches produced, defined as patches that successfully fix the bugs and pass all test cases. The greater the number of plausible patches generated, the better the model's performance. Figure 4 illustrates the overall performance comparison of both closed-source and open-source models within the standard and CoT prompting scenarios. DeepSeek-V3 and GPT-4o exhibit the best performance, with significant improvements noted under the CoT prompting method, while some models, such as o1-mini and Gemini-1.5-Pro experience a decline in performance. In terms of open-source model performance, DeepSeek-V3 and Llama-3.3-70B models demonstrate enhanced performance with CoT prompting, with DeepSeek-V3 achieving the highest results. Conversely, in the closed-source model performance, most models show improved performance with CoT prompting, such as GPT-4o and Grok-2.

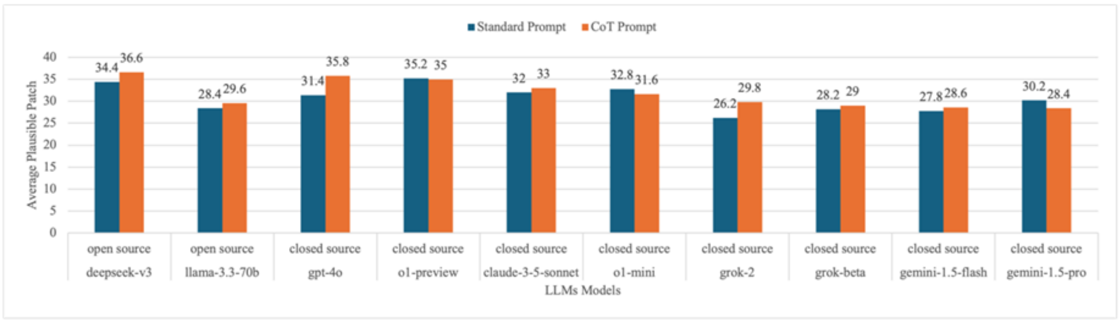


Figure 4. The comparison of the number of plausible patches using LLM models

The performance test results indicate that the use of CoT prompting generally enhances the performance of LLM models for APR tasks. It also reveals substantial performance differences among the LLM models, especially regarding the number of plausible patches generated and the cost efficiency of token usage. Closed-source models exhibit varied outcomes in APR tasks with CoT prompting. Most models, such as GPT-4o and Grok-2, experience significant performance improvements. For instance, GPT-4o records an increase from 31.4 to 35.8 plausible patches, while Grok-2 rises from 26.2 to 29.8 plausible patches. This improvement suggests that CoT prompting successfully guides models in tackling reasoning tasks more structurally, as explained by White *et al.* [8].

However, not all models display positive results with CoT prompting. Some models, such as o1-mini and o1-preview, show a decline in performance. A primary reason for this is the internal CoT characteristic inherent to these models. As described in the OpenAI documentation [28], the o1 models are designed to perform complex reasoning independently using internal mechanisms before arriving at an answer. This capability allows the model to solve complex problems without needing additional guidance from the prompt. Since the CoT prompting is applied to models like o1, the step-by-step instructions provided in the prompt might disrupt the model's already optimized internal reasoning process. This misalignment leads to decreased performance because the prompt structure does not align with the model's native reasoning approach.

However, open-source models such as DeepSeek-V3 and Llama-3.3-70B demonstrate significant performance differences with CoT prompting. The possible reason is the DeepSeek-V3 has a very large model size with a total of 671 billion parameters, whereas Llama-3.3-70B has only 70 billion parameters. The superior performance of DeepSeek-V3 aligns with the findings of Yu *et al.* [29], who noted that larger models tend to exhibit significant performance leaps with CoT prompting. This is because larger models have a greater capacity to understand and solve complex problems. In addition, training data also plays a substantial role in DeepSeek-V3's performance, which is trained using 14.8 trillion tokens from diverse and high-quality data [17]. Such rich training data endows the model with broad knowledge and enhanced reasoning capabilities, particularly in tasks involving CoT prompting. Yu *et al*. [29] assert that training data covering a wide range of reasoning materials and relevant knowledge can significantly influence a model's reasoning ability in CoT prompting tasks.

## 3.2. Token usage costs

Figure 5 presents a comparison of token usage costs for all LLM models. DeepSeek-V3 and Gemini-1.5-Flash record the lowest costs, while o1-preview incurs the highest costs for both methods. Generally, CoT prompting increases token usage costs, although certain models like o1-mini experience a cost reduction. The results showed that DeepSeek-V3 records the most efficient costs in both scenarios, while Llama-3.3-70B shows a slight cost increase under CoT prompting for open-source models. Conversely, in the comparison of token usage costs for closed-source models, models like Gemini-1.5-Flash have the lowest costs for both methods, whereas o1-preview records the highest costs. Although CoT prompting generally increases token costs, certain models, such as o1-mini, experience a decrease in costs. This increase can be attributed to the nature of CoT prompting, which requires a more extended, step-by-step reasoning process, resulting in more detailed output and greater token usage, as noted by Kojima *et al*. [21]. However, in models like o1-mini, CoT prompting reduces token costs because the prompt design limits the reasoning steps to five specific steps, thereby producing more concise responses compared to standard prompts that allow the model full freedom. In detail, open-source models like DeepSeek-V3 record the most cost-efficient performance with only \$0.006 for CoT prompting. Conversely, models such as o1-preview incur the highest costs with both methods, reaching \$6.775 with CoT prompting. This analysis underscores that token cost efficiency depends not only on the prompting method but also on the model's architectural design. Furthermore, the report from Artificial Analysis also mentioned that there were three highlights about a LLM model, such as the intelligence, speed, and price [16]. Hence, the practical development of APR in the educational or commercial usages should be aware about price or token usage cost since it requests to the closed source LLM model from paid API endpoint or cost to run the open source LLM model in the server with graphics processing unit (GPU).

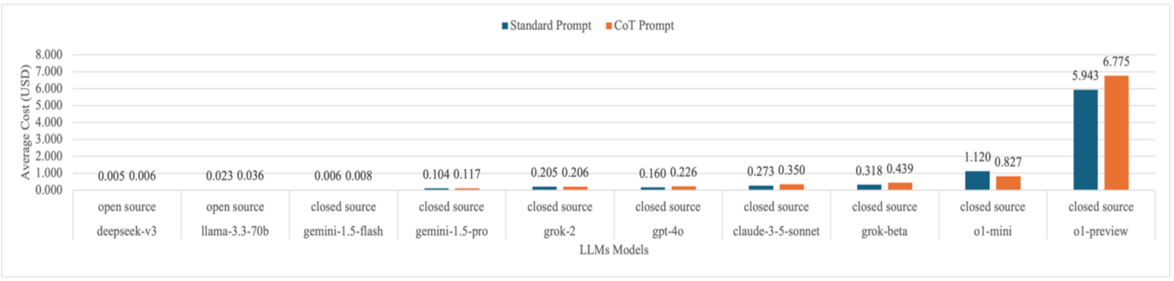
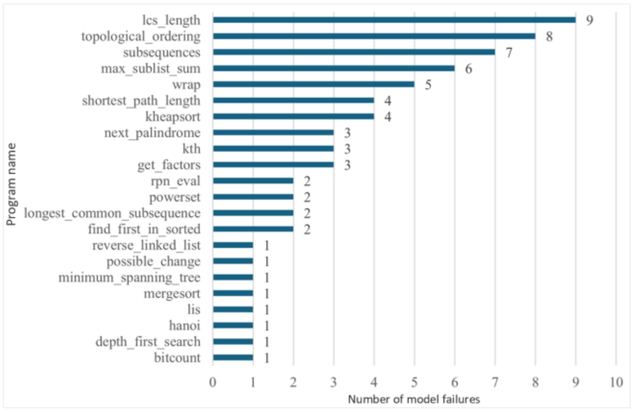


Figure 5. The comparison of the token usage cost

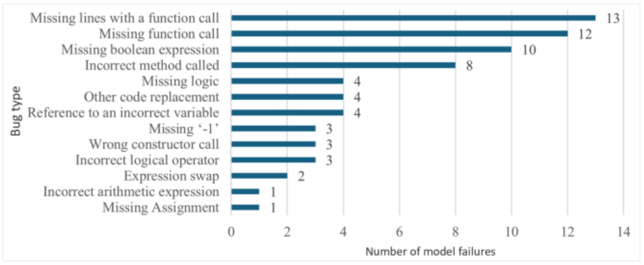
## 3.3. Identification of failure patterns

The data presented in Table 2 (see in Appendix) are derived from the highest performance test results for each model, based on five tests conducted using CoT prompting. The highest performance is determined by the number of plausible patches successfully generated. Each program in the QuixBugs benchmark is evaluated, with a checkmark (✓) indicating the model's success in fixing a bug and a cross (✗) indicating failure. This value represents the total number of bugs successfully fixed by the model in QuixBugs, based on the plausible patches produced. For example, the DeepSeek-V3 model recorded a total of 36.6 plausible patches in average or 37 failure patterns in total, which is the highest number of bugs successfully fixed by this model. Furthermore, based on Table 2, the distribution of model failures based on the types of bugs identified in the QuixBugs benchmark is illustrated in Figure 6. The number of failures for each program is calculated by summing the model failures for that specific program. The program with the highest number of failures is "lcs_length" (9 failures), indicating that this program is among the most challenging for the models to repair, as shown in Figure 6(a). On the other hand, the number of failures for each type of bug is calculated by summing the failures across all programs that contain the same type of bug. The bug types with the highest number of failures are "Missing lines with a function call" (13 failures) and "Missing function call" (12 failures), as shown in Figure 6(b).

These results align with the findings of Ye *et al*. [15], who identified that fixing these bugs requires a more complex approach compared to other bug types. The "Missing lines with a function call" bug necessitates that the model not only detects the bug's location but also inserts one or more new lines of code. Meanwhile, the "Missing function call" bug requires the model to call a specific function that may not be readily available in the code context. The program "lcs_length" exhibits the highest failure rate, with 9 out of 10 models failing to resolve it. Ye *et al*. [15] noted that this program contains two bugs, such as "Incorrect array slice" and "Missing boolean expression." This program is challenging to fix because it involves a multi-location patch, requiring changes in several interdependent parts of the code. It is difficult not only to detect but also to repair simultaneously. As highlighted by Ye *et al*. [15], multi-location and multi-bug patches present significant challenges in APR research.



(a)



(b)

Figure 6. The detailed distribution of model failures based on: (a) the program name and (b) the bug type

## 4. CONCLUSION

This study successfully developed an effective CoT prompting structure to enhance the performance of LLMs in APR tasks. The designed CoT prompting structure has been generally shown to improve the ability of LLM models to generate solutions for APR tasks. The DeepSeek-V3 model recorded the highest performance with an average of 36.6 plausible patches in the CoT prompting scenario, followed by GPT-4o with an average of 35.8 plausible patches. Additionally, DeepSeek-V3 demonstrated the best cost efficiency. These results confirm that CoT prompting is an effective method for improving the accuracy and quality of solutions generated by LLMs. The limitations of this study include the use of a dataset restricted to only the Python version of the QuixBugs dataset, as well as the use of LLM models that are only publicly accessible through APIs. Future studies could explore the detailed qualitative analysis of CoT process in APR with more variants of LLMs model as well. One possible implementation is the development of AI-based platforms, such as interactive dashboards for coding education. In addition, the investigation of complex bugs such as multiple bugs in multi-files and the library updates isuses in the syntax codes could also address in the future study. It could achieve by using the API threads or batch mechanism to send the request at once rather than one-by-one request. Furthermore, the use of reterival-augmented generation (RAG) mechanism with CoT for APR can be explore in future study with more testing scenarios, metrics, and qualitative analysis as well.

## ACKNOWLEDGEMENTS

This research was supported by the Directorate of Research and Community Service, Telkom University and Research Center for Data and Information Sciences, BRIN. We also thanks for the comments and suggestions of the future study from the reviewers.

## FUNDING INFORMATION

This research was supported by Directorate of Research and Community Service, Telkom University.

## AUTHOR CONTRIBUTIONS STATEMENT

This journal uses the Contributor Roles Taxonomy (CRediT) to recognize individual author contributions, reduce authorship disputes, and facilitate collaboration.

| Name of Author | C | M | So | Va | Fo | I | R | D | O | E | Vi | Su | P | Fu |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Eko Darwiyanto | ✓ | ✓ |  | ✓ | ✓ | ✓ | ✓ |  |  | ✓ |  | ✓ | ✓ | ✓ |
| Rizky Akbar Gusnaen | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |  | ✓ | ✓ | ✓ | ✓ |  |  |  |
| Rio Nurtantyana | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |  |  |  | ✓ | ✓ | ✓ | ✓ | ✓ |

| | | | | | |
|---|---|---|---|---|---|
| C | : **C**onceptualization | I | : **I**nvestigation | Vi | : **Vi**sualization |
| M | : **M**ethodology | R | : **R**esources | Su | : **Su**pervision |
| So | : **So**ftware | D | : **D**ata Curation | P | : **P**roject administration |
| Va | : **Va**lidation | O | : Writing - **O**riginal Draft | Fu | : **Fu**nding acquisition |
| Fo | : **Fo**rmal analysis | E | : Writing - Review & **E**diting | | |

## CONFLICT OF INTEREST STATEMENT

Authors state no conflict of interest.

## DATA AVAILABILITY

Derived data supporting the findings of this study are available from the corresponding author, [RN], on request.

## REFERENCES

[1] H. Zhong and Z. Su, "An empirical study on real bug fixes," *International Conference on Software Engineering*, vol. 1, pp. 913–923, 2015, doi: 10.1109/ICSE.2015.101.

[2] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen, "Reversible debugging software: quantify the time and cost saved using reversible debuggers," *Student Educational Project*, University of Cambridge, Cambridge, England, 2013.

[3] K. Huang *et al.*, "A survey on automated program repair techniques," *arXiv:2303.18184*, 2023.

[4] K. Liu *et al.*, "A critical review on the evaluation of automated program repair systems," *Journal of Systems and Software*, vol. 171, 2021, doi: 10.1016/j.jss.2020.110817.

[5] J. F.-Ansley, P. Denny, A. L.-Reilly, E. A. Santos, J. Prather, and B. A. Becker, "My AI wants to know if this will be on the exam: testing OpenAI's codex on CS2 programming exercises," in *ACM International Conference Proceeding Series*, 2023, pp. 97–104, doi: 10.1145/3576123.3576134.

[6] J. F.-Ansley, P. Denny, B. A. Becker, A. L.-Reilly, and J. Prather, "The robots are coming: exploring the implications of OpenAI codex on introductory programming," in *ACM International Conference Proceeding Series*, 2022, pp. 10–19, doi: 10.1145/3511861.3511863.

[7] J. A. Prenner, H. Babii, and R. Robbes, "Can OpenAI's codex fix bugs?: an evaluation on QuixBugs," in *International Workshop on Automated Program Repair, APR 2022*, 2022, pp. 69–75, doi: 10.1145/3524459.3527351.

[8] J. White *et al.*, "A prompt pattern catalog to enhance prompt engineering with ChatGPT," *PLoP '23: Proceedings of the 30th Conference on Pattern Languages of Programs,* 2023, pp. 1-31.

[9] J. Wei *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in Neural Information Processing Systems*, vol. 35, 2022.

[10] Z. Al Nazi, M. R. Hossain, and F. Al Mamun, "Evaluation of open and closed-source LLMs for low-resource language with zero-shot, few-shot, and chain-of-thought prompting," *Natural Language Processing Journal*, vol. 10, 2025, doi: 10.1016/j.nlp.2024.100124.

[11] A. Lewkowycz *et al.*, "Solving quantitative reasoning problems with language models," *Advances in Neural Information Processing Systems*, vol. 35, 2022.

[12] M. Lu, F. Gao, X. Tang, and L. Chen, "Analysis and prediction in SCR experiments using GPT-4 with an effective chain-of-thought prompting strategy," *iScience*, vol. 27, no. 4, 2024, doi: 10.1016/j.isci.2024.109451.

[13] D. Lin, A. Chen, J. Koppel, and A. S.-Lezama, "QuixBugs: a multi-lingual program repair benchmark set based on the Quixey challenge," in *SPLASH Companion 2017 - Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, 2017, pp. 55–56, doi: 10.1145/3135932.3135941.

[14] Q. Zhang, C. Fang, Y. Ma, W. Sun, and Z. Chen, "A survey of learning-based automated program repair," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 2, 2023, doi: 10.1145/3631974.

[15] H. Ye, M. Martinez, T. Durieux, and M. Monperrus, "A comprehensive study of automatic program repair on the QuixBugs benchmark," *Journal of Systems and Software*, vol. 171, 2021, doi: 10.1016/j.jss.2020.110825.

[16] Artificial Analysis AI, "Artificial analysis AI review: 2024 highlights," *artificialanalysis.ai*, 2025. Accessed: Dec. 09, 2024, [Online]. Available: https://artificialanalysis.ai/downloads/ai-review/2024/Artificial-Analysis-AI-Review-2024-Highlights.pdf

[17] DeepSeek-AI *et al.*, "DeepSeek-V3 technical report," *arXiv:2412.19437*, 2025.

[18] Y. Zhu, J. Li, G. Li, Y. F. Zhao, Z. Jin, and H. Mei, "Hot or cold? adaptive temperature sampling for code generation with large language models," in *AAAI Conference on Artificial Intelligence*, 2024, vol. 38, no. 1, pp. 437–445, doi: 10.1609/aaai.v38i1.27798.

[19] G. G. Lee, E. Latif, X. Wu, N. Liu, and X. Zhai, "Applying large language models and chain-of-thought for automatic scoring," *Computers and Education: Artificial Intelligence*, vol. 6, 2024, doi: 10.1016/j.caeai.2024.100213.

[20] J. Sun, F. Li, X. Qi, H. Zhang, and J. Jiang, "Empirical evaluation of large language models in automated program repair," *arXiv:2506.13186*, 2025.

[21] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," *NIPS'22: Proceedings of the 36th International Conference on Neural Information Processing Systems*, vol. 17, pp. 302, Jan. 2023, doi: 10.5555/3600270.3601883.

[22] S. A. Asadollah, D. Sundmark, S. Eldh, H. Hansson, and W. Afzal, "10 years of research on debugging concurrent and multicore software: a systematic mapping study," *Software Quality Journal*, vol. 25, no. 1, pp. 49–82, 2017, doi: 10.1007/s11219-015-9301-7.

[23] Claude Docs, "Crafting effective examples," *docs.claude.com,* 2024. Accessed: Oct. 31, 2024, [Online]. Available: https://docs.claude.com/en/docs/build-with-claude/prompt-engineering/multishot-prompting#crafting-effective-examples

[24] Y. Li, "A practical survey on zero-shot prompt design for in-context learning," in *International Conference Recent Advances in Natural Language Processing, RANLP*, 2023, pp. 641–647, doi: 10.26615/978-954-452-092-2_069.

[25] B. Okken, *Python testing with pytest*, Second edi. Pragmatic Bookshelf, 2022.

[26] B. Yang and J. Yang, "Exploring the differences between plausible and correct patches at fine-grained level," in *IBF 2020 - 2020 IEEE 2nd International Workshop on Intelligent Bug Fixing*, 2020, pp. 1–8, doi: 10.1109/IBF50092.2020.9034821.

[27] R. Gharibi, M. H. Sadreddini, and S. M. Fakhrahmad, "T5APR: empowering automated program repair across languages through checkpoint ensemble," *Journal of Systems and Software*, vol. 214, 2024, doi: 10.1016/j.jss.2024.112083.

[28] OpenAI, "Learning to reason with LLMs," *openai.com,* 2025. Accessed: Jan. 15, 2025. [Online]. Available: https://openai.com/index/learning-to-reason-with-llms/.

[29] Z. Yu, L. He, Z. Wu, X. Dai, and J. Chen, "Towards better chain-of-thought prompting strategies: a survey," *arXiv:2310.04959*, 2023.

## APPENDIX

Table 2. The LLM models test results on various bugs in the QuixBugs benchmark

| Buggy program name | Bug type | GPT-4o | o1-Preview | o1-mini | Claude-3-5-Sonnet | Gemini-1.5-Flash | Gemini-1.5-Pro | Grok-Beta | Grok-2 | DeepSeek-V3 | Llama-3.3-70B | Total |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| bitcount | Incorrect logical operator | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | 1 |
| breadth_first_search | Missing boolean expression | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 0 |
| bucketsort | Reference to an incorrect variable | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 0 |
| depth_first_search | Missing lines with a function call | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | 1 |
| detect_cycle | Missing boolean expression | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 0 |
| find_first_in_sorted | Incorrect logical operator | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | 2 |
| find_in_sorted | Missing '+1' | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 0 |
| flatten | Missing function call | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 0 |
| gcd | Expression swap | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 0 |
| get_factors | Wrong constructor call | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | ✗ | 3 |
| hanoi | Reference to an incorrect variable | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | 1 |
| is_valid_parenthesization | Other code replacement | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 0 |
| kheapsort | Missing function call | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | ✓ | ✓ | 4 |

Table 2. The LLM models test results on various bugs in the QuixBugs benchmark *(continued)*

| Buggy program name | Bug type | GPT-4o | o1-Preview | o1-mini | Claude-3-5-Sonnet | Gemini-1.5-Flash | Gemini-1.5-Pro | Grok-Beta | Grok-2 | DeepSeek-V3 | Llama-3.3-70B | Total |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| knapsack | Incorrect comparison operator | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 0 |
| kth | Reference to an incorrect variable | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ | ✓ | ✓ | 3 |
| lcs_length | Incorrect array slice Missing boolean expression | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | 9 |
| levenshtein | Missing '+1' | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 0 |
| lis | Missing logic | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | 1 |
| longest_common_subsequence | Missing function call | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✗ | 2 |
| max_sublist_sum | Missing function call | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | ✓ | 6 |
| mergesort | Incorrect arithmetic expression | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | 1 |
| minimum_spanning_tree | Missing logic | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | 1 |
| next_palindrome | Missing '-1' | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ | ✓ | ✓ | ✓ | 3 |
| next_permutation | Incorrect comparison operator | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 0 |
| pascal | Missing '+1' | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 0 |
| possible_change | Missing boolean expression | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 1 |
| powerset | Missing logic | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | 2 |
| quicksort | Incorrect comparison operator | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 0 |
| reverse_linked_list | Missing assignment | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 1 |
| rpn_eval | Expression swap | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ | 2 |
| shortest_path_length | Other code replacement | ✗ | ✗ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✗ | 4 |
| shortest_path_lengths | Expression swap | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 0 |
| shortest_paths | Missing function call | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 0 |
| shunting_yard | Missing lines with a function call | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 0 |
| sieve | Incorrect method called | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 0 |
| sqrt | Incorrect arithmetic expression | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 0 |
| subsequences | Missing lines with a function call | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | 7 |
| to_base | Expression swap | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 0 |
| topological_ordering | Incorrect method called | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | 8 |
| wrap | Missing lines with a function call | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ | ✓ | 5 |
| Total | | 36 | 37 | 33 | 34 | 30 | 29 | 32 | 31 | 37 | 33 | - |

Note: The data represent the highest performance results for each mode that obtained from five test iterations using CoT prompting.

## BIOGRAPHIES OF AUTHORS

**Eko Darwiyanto** is currently working as a lecturer in the Informatics Study Program, School of Computing, Bandung, Indonesia. His expertise in software development methodologies and software engineering. He can be contacted at email: ekodarwiyanto@telkomuniversity.ac.id.

**Rizky Akbar Gusnaen** is currently a student at the Telkom University, Indonesia. He has a keen interest in software engineering, artificial intelligence, and blockchain. He can be contacted at email: riakgu@student.telkomuniversity.ac.id.

**Rio Nurtantyana** received the Ph.D. degree. He is currently working as a Researcher with the Research Center for Data and Information Sciences, National Research and Innovation Agency (BRIN), Bandung, Indonesia. He is also a lecturer in software engineering courses with the School of Computing, Telkom University, Bandung. His research interests include the IoT, AI, and mobile learning with AR/VR to transform the education in digital era. He was selected as an Honorary Member of the Phi Tau Phi Scholastic Honor Society of the Republic of China, in 2023. He can be contacted at email: akunerio@gmail.com.