Evaluating LLM-Generated Bug Reports Using CI Failures and Code Diffs

Team 8

Jun Wu
jlnwu@ucdavis.edu
University of California, Davis
Davis, CA, USA

Ahnaf Faisal
ahffaisal@ucdavis.edu
University of California, Davis
Davis, CA, USA

Chifang Chou
cfchou@ucdavis.edu
University of California, Davis
Davis, CA, USA

Jintian Xu
jttxu@ucdavis.edu
University of California, Davis
Davis, CA, USA

Abstract

CI systems generate large volumes of build logs that contain essential debugging information, but manually interpreting these logs and transforming them into high-quality bug reports remains labor-intensive for developers. With the rapid advancement of LLMs, recent research has begun to explore whether these models can automatically produce accurate and useful bug reports directly from raw build log outputs. This project evaluates the effectiveness of several LLMs (models from OpenAI and Gemini) in generating bug reports using real-world failing artifacts from the BugSwarm dataset. For each artifact, we collected CI build logs and code diffs, and applied two prompting strategies (simple and smart) to investigate how prompt design affects bug-report quality. We manually scored the generated reports using a rubric based on prior software engineering literature, assessing key-aspect completeness, code correctness, stack trace precision, patch accuracy, readability, and structural clarity. Our results show that LLMs are capable of generating well-structured and largely accurate bug reports, with smart prompts consistently improving report clarity and completeness. However, LLMs still demonstrate occasional hallucinations and misinterpretations, especially when prompts provide insufficient context. These findings highlight both the promise and limitations of LLM-based bug report generation, offering practical insights for integrating LLMs into developer workflows.

1 Introduction

Modern software development relies heavily on CI systems to automatically detect build failures, test regression, and configuration issues. Although CI logs often contain detailed technical information, manually interpreting build failures and transforming them into clear and actionable bug reports remains a time-consuming task for developers. With the rapid advancement of LLM, there is growing interest in whether these models can automatically generate high-quality bug reports using only raw CI build logs and code diffs.

This project investigates the effectiveness of LLMs, including OpenAI and Gemini variants in generating bug reports from real-world software artifacts drawn from the BugSwarm dataset. For each failing artifact, we extract the build logs, gather the corresponding code diff, and then prompt different LLMs using two distinct prompting strategies, "simple" and "smart". The resulting bug reports are evaluated using a custom rubric that measures correctness, completeness, clarity and technical accuracy.

By systematically analyzing LLM-generated bug reports across models, prompts, and artifact types, this study aims to understand the capabilities and limitations of modern LLMs that can serve as reliable assistants for developers who need to interpret CI failures and maintain software quality in large, evolving codebases.

2 Research Questions

- Can LLMs reliably generate accurate bug reports from build logs and code diffs?
  This question examines whether the bug reports produced by different LLMs correctly capture the root cause of build failures, without introducing hallucinations or incorrect technical details.
- How do different LLM models compare in bug-report quality?
  We listed out the evaluation criterias and compared models from OpenAI and Gemini across multiple dimensions including precision, clarity, conciseness, and technical correctness to determine whether certain architectures consistently outperform others.
- Does prompt design influence the quality of LLM-generated bug reports?
  We evaluate whether "smart" prompts produce better reports than minimal "simple" prompts, and analyze

which prompts help LLMs extract relevant failure information.

3 Implementation

3.1 Pipeline

Here is the pipeline for generating and evaluating a bug report using LLM

- Extract the build log (failed version) and code diffs of an artifact tag using the BugSwarm API.
- Input the build log + code diff to the LLMs using their APIs.
- LLM generates a bug report
- Evaluate the quality of the bug report using Grading Metries in Section 3.5

3.2 Models Used

We used two paid models from Google Gemini and OpenAI. The Gemini model was 'gemini-2.5-flash' and the OpenAI model was 'gpt-5-mini'. The integration for both of them was quite similar. We first fetch the build log and code diff using the Bugswarm api and then send these logs to the model alongside our prompt and save the response(report) we get from them. The only difference between the two models was that we could use direct files (build log and code diff) to upload to the Gemini model. Unfortunately, we could not do the same for OpenAI model and had to use a plain string to give it the build log and code diff.

3.3 Prompt Differences

For both the models, we tried two different prompting methods. Firstly, we tried the zero shot prompting with no detailed instructions and simply just asked the model to generate a bug report based on the build log and code diff given to it. Secondly, we tried the structured instruction prompting which had clear instructions on what the report should contain.

3.4 Readability

For computing readability score, we first used a tool called TextDescriptives[6]. It is a Python library built on top of spaCy[7]. It computes a wide range of text metrics such as Flesch Reading Ease[8], Flesch–Kincaid Grade Level etc. It also gives some surface text statistics like sentence count, word count etc. In our work, we use this tool to save the Flesch Reading Ease score for each report. This score estimates how easy a text is to read. The formula used for estimating this score is:

$$\text{FRE} = 206.835 - 1.015 \cdot \left(\frac{\text{total words}}{\text{total sentences}} \right) - 84.6 \cdot \left(\frac{\text{total syllables}}{\text{total words}} \right)$$

TestDescriptives internally calculates the number of sentences and words using spaCy and then uses this formula to calculate the score. The scores are generally expected to be in between 0-100, although in very few cases we also encountered negative scores. The score range 90-100 means very easy to read, 60-70 means standard and lower than 30 means very difficult to read and could be academically complex. In our bug reports, a Flesch Reading Ease score of around 60–70 generally indicated that the LLM produced reports that were easy to read. We did not directly use this FRE score provided by TextDescriptives. Rather we scaled the scores between 0-2 depending on the scores all the reports in our dataset got. The formula looked something like this:

$$\text{scaled score} = \frac{x - \min}{\max - \min}$$

After that depending on the range we gave the final score between 0-2. We used equal width buckets:

Simple Prompt

```
 The uploaded file contains TWO sections:
1. BUILD LOG
2. CODE DIFF

Both are required for full debugging context.

Create a detailed software bug report with these
information.
```

Smart Prompt

```
 The uploaded file contains TWO sections:
1. BUILD LOG
2. CODE DIFF

Both are required for full debugging context.

You are a software engineer assistant. Based on the
build log and code diff, generate a detailed bug report
that includes:
1. Title: Provide a concise title describing the bug.
2. Steps to Reproduce: List clear, itemized steps.
3. Triggering Input: Include the input, parameters, or
conditions that caused the bug.
4. Expected Behavior: Describe what the program should
do.
5. Observed Behavior: Describe what actually happens,
including error messages.
6. Relevant Code Snippets: Include relevant code from
the diff.
7. Stack Traces: Include necessary stack trace from
the build log.
8. Patches / Suggested Fixes: Include if available in
the diff.
```

Figure 1. Simple vs Smart Prompt Templates

3.5 Grading Metrics

To systematically evaluate the quality of the generated bug reports, we designed a set of metrics based on prior empirical studies of bug report effectiveness [1]. High-quality bug reports are characterized by the presence of relevant technical

range	Final score
0.00 - 0.19	0
0.20 - 0.39	0.5
0.40 - 0.59	1
0.60 - 0.79	1.5
0.80 -1.00	2

Table 1. Ranges and their corresponding final score

information and clear organization, which directly supports program developers in reproducing and solving issues.

Our feature metrics include key-aspect completeness, code samples, stack traces, patches, readability and structure. For each feature, we assigned a score based on the quality of the corresponding part of the bug report, resulting in a final score out of 10 for each report. The specific criteria are listed as follows:

Key-Aspect Completeness (Total score: 4). We assessed whether the bug report includes the following 4 components, awarding 1 point for each included component:

- Triggering information (Score: 0 / 1):
  Includes the relevant methods and the triggering conditions or parameters. Such information can help developers understand under what circumstances the bug occurs.
- Expected behavior(Score: 0 / 1):
  Describes the intended behavior of the program, providing a reference for identifying the deviations.
- Observed behavior(Score: 0 / 1):
  Describes the actual behavior of the program, allowing developers to pinpoint the issue accurately.
- Steps to reproduce(Score: 0 / 1):
  Lists the procedures needed to trigger the bug. This enables the developers to reproduce the issue reliably.

Code samples (Score: 0 / 0.5 / 1). We examined whether the bug report includes the relevant code snippet where the bug occurs, which facilitates understanding in context. If no code sample is included, the score is 0. If a code sample is included, and it matches the build log and code diff we provided to the LLM exactly, the score is 1. If a code sample is included, but it does not fully match the provided files, the score is assigned to 0.5.

Stack Traces (Score: 0 / 0.5 / 1). Stack traces typically consist of a start line and trace lines. We evaluated whether the bug report contains these critical lines. A complete and focused stack trace that includes all essential lines necessary for debugging earns 1 point. If the stack trace is a mechanical copy of the entire build log, the score is 0.5, as excessively long traces can obscure relevant information, making it harder for developers to identify the root cause during debugging.

Patches (Score: 0 / 1). A description of patch or fix is evaluated based on the correctness compared to the code diff on the BugSwarm website. Correct patches receive 1 point, whereas incorrect or missing patches receive 0 points.

Readability (Score: 0 / 0.5 / 1 / 1.5 / 2). We use the readability tool introduced in section 3.4 to quantify the clarity and comprehensibility of the bug report. The raw readability values are min-max normalized across all reports, and the normalized value is then discretized into five levels: 0, 0.5, 1, 1.5, and 2, based on fixed percentile thresholds.

Structure (Score: 0 / 1). This metric evaluates the overall organization of the report, including the presence of itemization and indentation, which improve the report's accessibility for human readers. Reports that present information in a clear, structured, and well-separated manner receive 1 point, whereas reports in which all content is densely packed without meaningful structure receive 0 points.

4 Results and Observations

4.1 Statistical Analysis

Models	Gemini 2.5-flash		GPT 5-mini	
Prompt	Simple	Smart	Simple	Smart
Mean	7.76	8.93	6.78	8.58
Median	7.75	9.00	7.00	8.75

Table 2. Statistics Analysis for LLM-Generated Bug Reports

We generate bug reports for 40 Java artifacts. For each artifact, we generated 4 reports using combinations: Gemini + "simple" prompt, Gemini + "smart" prompt, GPT + "simple" prompt, and GPT + "smart" prompt. After that, we manually scored all the reports using the grading metrics in Section 4.4. After all the reports were scored, we computed a statistical analysis on the scores of the bug reports.

Based on the mean and median values in Table 1, the Gemini model performs better than the GPT 5 model, in terms of both the "simple" prompt and the "smart" prompt. For the "simple" prompt, Gemini scored 1 point more than the GPT on average, while for the "smart" prompt, Gemini scored slightly higher by approximately 0.3 points. Based on the results in Table 3, Gemini generally does better in areas such as Expected behaviors, Step to reproduce, and Readability. Even though Gemini got higher scores in this case, we cannot conclude that Gemini is a better model compared to GPT, because the performance of the models depends on many factors. All we can say is that Gemini performs better under our custom grading metrics.

Based on the results in Table 2, the "smart" prompting gives higher scores compared to the "simple" prompting. The scores of the "smart" prompt are approximately 1.2 to 1.8 higher. This is as expected, because we explicitly instruct the models on the desired sections to include in the bug report, so they will generate bug reports that are more aligned

with our grading metrics. Based on Table 3 and our observations, reports generated using "simple" prompts sometimes do not include or do not give enough amount of information (to determine the bug) of Triggering input, Code samples, and Stack trace. Those are the main areas that cause the "simple" prompt to have lower scores. Ultimately, the scores of both "simple" and "smart" are close (only 1.2 to 1.8 different), which indicates that the models are "smart" enough to generate a report that contains most of the important information, without explicitly prompting them to do so.

Based on Table 2, the best-performing combination is Gemini + "smart" prompt, and based on Table 3, it gets perfect or almost perfect scores for every criterion. The worst-performing combination is GPT + "simple" prompt. Based on our observations, this combination usually does not include expected behavior in the reports, while other combinations do. Also, GPT + "simple" prompt tends to explain the patches in words rather than give code examples, which causes its Code Sample score to be lower compared to Gemini.

Models & prompts	Mean - Expected behaviors (0/1)	Mean - Observed behaviors (0/1)	Mean - Triggering Input (0/1)	Mean - Step to reproduce (0/1)	Mean - Code Samples (0/0.5/1)	Mean - Stack Traces (0/0.5/1)	Mean - Patches (0/1)	Mean - Readability (0–2)	Mean - Structure (0/1)
Gemini Simple Prompt	0.98	0.98	0.63	1.00	0.46	0.45	0.63	1.65	1.00
Gemini Smart Prompt	1.00	1.00	1.00	1.00	0.83	0.85	0.86	1.39	1.00
OpenAI Simple Prompt	0.45	0.93	0.78	0.68	0.31	0.39	0.83	1.43	1.00
OpenAI Smart Prompt	1.00	1.00	0.95	1.00	0.79	0.75	0.83	1.26	1.00
Total	**0.86**	**0.98**	**0.84**	**0.92**	**0.60**	**0.61**	**0.78**	**1.43**	**1.00**

Table 3. Mean Score for Each Criterion in Grading Metrics

4.2 Other Findings

1. *Well structured reports.*
   A first intuitive observation is that all evaluated LLMs consistently produced well-structured bug reports, even though the prompts did not explicitly include any formatting guidance. This behavior is likely due to the inherent training of modern LLMs on large corpora of technical documentation, issue reports, and instruction-following data, leading them to organize information in a clean and systematic manner. Such structural coherence is beneficial in bug reporting, since a well-formatted report enables developers to more quickly locate key information, and consequently accelerates the debugging process.
2. *More detailed prompts yield more targeted and discoverable reports.*
   We observe that the smart prompts produce bug reports in which relevant information is explicitly enumerated as separate items or bullet points, making it easier for developers to locate specific facts. By contrast, in reports generated from simple prompts, the information is presented, but scattered rather than organized into discrete, findable units. Consequently, prompt specificity improves the retrievability of critical debugging information.
3. *Prompt specificity can also suppress extraneous information.*
   An interesting pattern is that many reports produced from simple prompts include environment information (extracted from the build log) by default. In contrast, bug reports generated from smart prompts usually omit this, as the prompt did not explicitly ask for it. Thus, while smart prompts improve focus and clarity, they also make the generated report highly dependent on what the prompt specifies; omissions in the prompt can lead to omissions in the report. This trade-off implies that we must carefully design the prompt, and must balance concision with giving LLM the autonomy to include auxiliary details (e.g., environment, configuration) when appropriate.
4. *LLM can make mistakes.*
   In one case (artifact tag: `square-javapoet-69618779`), according to the code patches, the developer had deliberately annotated a failing test with `@Ignore` to skip the error. However, when given a simple prompt, Gemini misinterpreted the build log and concluded that the bug was caused by the developer's use of `@Ignore` in the passed version, rather than the actual issue involving an incorrect qualified type. This example illustrates that insufficient prompt context can cause LLMs to fill in missing details with inaccurate assumptions.
5. *Readability Scores.*
   In this project, we used TextDescriptives to automatically evaluate the readability of the report. From Table 3, the mean value of readability scores is 1.43, which is not the worst, but also not the best. This surprised us, because during our manual evaluation of the reports, all the reports were well structured and easy to read. This is likely due to TextDescriptives was mainly developed to evaluate natural language, not technical reports. The technical report usually contains code, file paths, and sentences that are not grammatically

correct. The tool might consider those to be bad grammar/writing.

5 Conclusion

This study examined the ability of modern LLMs to generate high-quality bug reports directly from build logs and code diffs, focusing on models from OpenAI and Gemini under two distinct prompting strategies. Across 40 real-world failing BugSwarm artifacts, our evaluation reveals several important observations. First, all evaluated LLMs reliably produced well-structured bug reports, even without explicit formatting instructions. This suggests that structural organization has effectively become an emergent capability of current LLMs, making them suitable for reporting tasks that require clarity and organization. Second, prompt design plays a critical role in determining report quality. Smart prompts led to clearer, more complete reports by guiding models to enumerate key debugging information such as root cause, expected/observed behavior, and steps to reproduce. In contrast, simple prompts produced more diffuse reports and occasionally encouraged models to include extraneous environment details. Despite these strengths, the models exhibited notable limitations. In several instances, LLMs misinterpreted the root cause or introduced unsupported explanations, particularly when the prompt lacked sufficient technical constraints. These errors highlight that while LLMs can assist in triaging CI failures, they cannot yet be treated as fully reliable autonomous agents without human verification. Overall, our results suggest that LLMs can substantially reduce developers' effort in interpreting CI failures by providing structured and context-rich bug reports, especially when paired with carefully engineered prompts. However, achieving consistently accurate technical reasoning remains an open challenge. Future work could explore automated prompt refinement, hybrid approaches that combine static analysis with LLM reasoning, and training domain-specific models optimized for CI-debugging scenarios.

6 Contributions

- **Ahnaf Faisal:** Integration of APIs and report generation, evaluation of 10/40 artifacts, Dockerization, final report writing (3.2-3.4).
- **Jun Wu:** Extract artifact names and setting up API, evaluation of 10/40 artifacts, creation of README, final report writing (4.1).
- **Chifang Chou:** Metrics design, evaluation of 10/40 artifacts, fianl report writing (Abstract, 1, 2, 5), final report organization and layout revision.
- **Jintian Xu:** Prompt design, metrics design, evaluation of 10/40 artifacts, final report writing (3.5, 4.2-1 4).

7 Project Repository

All source code, data processing scripts, and evaluation bug reports used in this project are available at:

https://github.com/ecs245/f25-team-8

References

[1] Bettenburg, N., Just, S., Schröter, A., Weiss, C., Premraj, R., and Zimmermann, T. *What makes a good bug report?* In Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 308–318, 2008. https://doi.org/10.1145/1453101.1453146

[2] Acharyya, J., and Ginde, G. *Can we enhance bug report quality using LLMs? An empirical study of LLM-based bug report generation.* arXiv preprint arXiv:2504.18804, 2025. https://arxiv.org/pdf/2504.18804

[3] Textstat Contributors. *textstat readability library.* Retrieved from https://github.com/textstat/textstat

[4] Kang, H., Park, Y. J., Lee, J., Ham, G., and Kim, S. *Libro: A benchmark for evaluating LLM-based bug report generation.* COINSE Technical Report, 2023. https://coinse.github.io/publications/pdfs/Kang2023aa.pdf

[5] COINSE Lab. *Libro: LLM-based bug report generation platform.* GitHub repository. https://github.com/coinse/libro

[6] Lundberg, M., Wolff, T., and Lilliecreutz, D. *TextDescriptives.* GitHub Repository, 2023. https://github.com/HLasse/TextDescriptives

[7] Honnibal, M., Montani, I., Van Landeghem, S., and Boyd, A. *spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing.* In Proceedings of the spaCy Workshop, 2020.

[8] Flesch, R. *A new readability yardstick.* Journal of Applied Psychology, 32(3), 221–233, 1948.