DocuRepoBench: Evaluating Documentation-Driven Repository-Level Code Generation

Stephanie Cho **Sarah Friedrichs** **Elaheh Rasoulian** **Julie Young**

Abstract

This study investigates the capability of large language models (LLMs) to perform repository-level code generation—a task that requires reasoning across interdependent files, documentation, and configurations. Unlike single-file generation, repository-level patching demands global consistency and awareness of project-wide design patterns. To address this challenge, we propose a two-stage pipeline in which one LLM ("documenter") summarizes the repository into structured documentation, and another ("coder") generates patches using this distilled context. Using the FEA-Bench Oracle-Lite dataset, we evaluate multiple Gemini and DeepSeek configurations with diff-based, similarity-based, and AI-as-a-Judge evaluations. While baseline models achieved higher patch-application success, the AI-judge assessment rated a Deepseek Documenter, Gemini Coder pipeline higher in quality, suggesting that structured context can enhance conceptual understanding even when exact formatting fails. These findings reveal a trade-off between syntactic precision and semantic reasoning and underscore the need for better intermediate representations to support repository-scale code generation.

1 Introduction

1.1 Motivation

This project aims to evaluate the performance of large language models (LLMs) in repository-level code generation, a setting that remains difficult for current models. Code generation represents one of the most impactful frontiers of applied AI, with direct implications for software engineering productivity, code maintenance, and system modernization. As software systems continue to grow in scale and complexity, the ability for models to generate and modify code across entire repositories rather than isolated files would enable more autonomous feature implementation, large-scale refactoring, and efficient migration of legacy systems. Understanding how LLMs perform in this setting therefore provides both scientific insight into model reasoning and practical value for real-world development workflows.

Unlike single-file tasks, repository-level patching requires models to reason over many interdependent files, follow project-wide design patterns, and integrate information scattered across documentation, configuration files, and import structures. These requirements frequently exceed the model's effective context window and challenge its ability to maintain global consistency across a codebase.

To address these challenges, we propose a two-stage pipeline: one LLM first analyzes the repository and produces structured summaries or documentation, and a second LLM uses this distilled context to generate or complete code. This design tests whether well-organized, context-aware information helps models reason more effectively than processing raw, unstructured source code directly. We additionally explored diff-based evaluation as an auxiliary perspective to understand how close model-generated patches are to the intended changes. These analyses motivated our interest in structured context, since even small inaccuracies in the diff often propagated into test failures, reinforcing the need for better contextual reasoning.

1.2 Data

All data in this study are collected from publicly available GitHub repositories using the official FEA-Bench pipeline Li et al. (2025), which retrieves repository metadata, commits, and source files via authenticated API access. The FEA-Bench pipeline and data is publicly available on GitHub[1]. We use the Oracle-Lite dataset variant of FEA-Bench, which provides a balanced level of con-

[1] https://github.com/microsoft/FEA-Bench

textual information—including target source files, relevant dependencies, and documentation snippets—while remaining computationally efficient for large-scale evaluation. This choice allows for consistent benchmarking across repositories and models without requiring full repository reconstruction.

1.3 Literature Review

One of the primary papers that has inspired our proposal is FEA-Bench (Li et al. (2025)), which provides a benchmark for evaluating code generation in repositories. The paper collects pull requests from GitHub Python repos spanning 1,401 total tasks. This work expands upon SWE-Bench (Jimenez et al. (2024)) covering 2,294 Python software problems. However, FEA-Bench primarily focuses on feature creation tasks rather than bug fixes. Importantly, both papers found that existing code generation models perform poorly in complicated code settings. In fact, increasing the amount of context actually decreased performance of the models, indicating that existing models are not good at pinpointing problematic code Jimenez et al. (2024) and may require more precise context retrieval Li et al. (2025).

MRG-Bench Li (2025) evaluates model performance across full software repositories including Python, Java, and Go projects by testing whether generated code passes executable unit tests. The results reveal that existing models achieve under 40% accuracy even with extended context, suggesting that simply providing longer input windows or retrieval-augmented information does not guarantee better understanding. Instead, performance depends heavily on the model's ability to infer task intent and reason about code functionality from meaningful documentation and structured context.

LongCodeBenchRando et al. (2025) also used large context windows to evaluate code and had a similar result, noticing that the larger the context window the worse the performance. They found performance drops such as 29% to 3% for Claude 3.5 Sonnet and 70.2% to 40% for Qwen 2.5.A³-CodGenLiao et al. (2024) introduces a context-aware code-generation framework that helps large language models reason about repository-level information. The authors demonstrate that existing models like ChatGPT and GitHub Copilot often ignore local, global, and third-party dependencies, leading to redundant or incompatible code. Their framework systematically extracts these three layers of context, including functions and variables from the current file, reusable modules from other files, and available external libraries, and fuses them into structured prompts for code generation. Experiments on their RepoEval benchmark show substantial gains in precision and F1 for code reuse and correctness compared with vanilla LLMs. This work highlights that structured, context-rich input can significantly improve functional accuracy, supporting our project's hypothesis that summarizing repositories into concise, well-organized documentation may enable LLMs to reason more effectively than direct long-context prompting.

From these insights, we identify two key gaps in prior research: (1) most studies rely on a single model that directly processes large, unstructured repositories, and (2) few explore systematic ways to construct structured context before code generation. To address this, we propose a two-step LLM pipeline—one model summarizes the repository into structured documentation, and another uses it for code generation.

2 Methods

To generate natural language documentation for each FEA-Bench task, we built a standalone "documenter" pipeline that consumes the oracle lite metadata and produces a structured feature-implementation guide for downstream patch-generation (See Figure 1 below for visualization) The script begins by parsing user arguments to select the LLM backend (Gemini or DeepSeek) and verifies that the appropriate API keys are available. It then loads the set of oracle tasks, filtered to only those instances whose reference patch successfully passes all tests, ensuring that documentation is generated only for valid repair scenarios. For each task, the system constructs a detailed documenter prompt by synthesizing the repository context, base commit metadata, and natural-language feature description. This prompt is combined with a fixed system instruction that specifies the expected structure of the output (e.g., repository overview, implementation plan, pseudo-code, and coding-style guidance). The selected LLM is then invoked to produce a full implementation document, which is saved to disk. This documenter pipeline serves as the first stage of our two-step approach, providing the coder model with a task-specific specification to guide patch generation.

To generate executable bug-fix patches from the

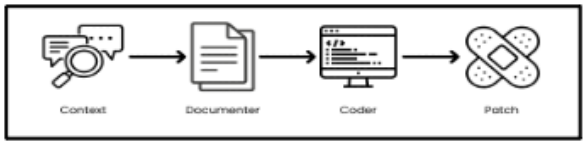

Figure 1: Figure 1: Documenter-Coder Pipeline

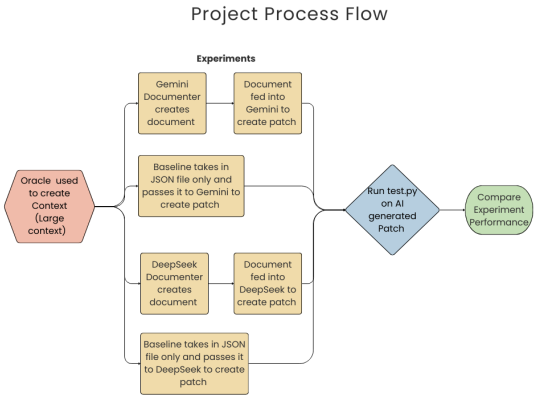

Figure 2: Figure 2: CAI Process Flow

documenter outputs, we implemented a second-stage "coder" pipeline that transforms the natural language design documents into unified-diff patches suitable for git apply (See Figure 2 Below for full process flow pipeline). The script begins by selecting the LLM backend (DeepSeek or Gemini) and verifying that the required API keys are available. It then loads the full set of oracle tasks—filtered to those with successful reference repairs—to ensure evaluation is performed only on valid bug-fixing instances. For each task, the coder retrieves the corresponding document produced by the documenter stage and constructs a composite prompt consisting of (1) the design document, which summarizes the intended behavior and relevant repository context, and (2) a strict system instruction that enforces the exact git-diff format required by FEA-Bench. This instruction guides the model to infer the necessary behavioral change, identify the appropriate files and functions to modify, and produce a minimal, style-consistent patch without supplying any reasoning text or unit tests. The selected LLM is then executed to generate the patch. This coder pipeline constitutes the second stage of our two-step architecture, turning high-level documentation into precise code modifications.

3 Experiments

We performed multiple experiments focusing on 20 repositories and 20 tasks. To evaluate how different forms of context influence repository-level patch generation, we designed eight model configurations spanning three experimental families. First, the baseline models provide a lower-bound comparison by supplying only the short JSON task specification from FEA-Bench, with patch generation performed by either DeepSeek or Gemini. These settings test how well each model performs without additional contextual reasoning. Next, the Documenter→Coder pipelines assess whether supplying structured repository summaries improves model performance: one LLM (either DeepSeek or Gemini) generates large-context documentation, and a second LLM uses this distilled information to produce patches. This evaluates whether organizing the repository into a structured summary reduces cognitive load and aids the coder model. We also include Documenter + Baseline hybrid configurations, which combine both the structured documentation and the original JSON specification to test whether blending high-level and task-local signals yields better results. Finally, to study cross-model generalization, we evaluate mixed pipelines where documentation produced by one model (e.g., Gemini) is used by another model (e.g., DeepSeek) for patch generation. Together, these eight configurations allow us to isolate the effects of context size, documentation quality, and model pairing on repository-level code generation. The Table 1 summarizes the set ups of our experiments.

We also refined our prompts as we performed these experiments. We experimented with role-based prompting, assigning the coder the role of a software engineer and the documenter the role of a technical writer. In addition, we incorporated hidden chain-of-thought (CoT) prompting, which guides the model to internally structure its reasoning without exposing intermediate steps in the final output. This technique allows the model to decompose patch-generation tasks into smaller reasoning units—such as identifying the target file, isolating the faulty logic, and mapping the expected fix to the diff format—before producing the final patch. By encouraging more deliberate internal reasoning, hidden CoT reduced hallucinated edits and improved alignment between the required change and the generated modification.

We also included a one-shot example demon-

strating the correct git diff structure, which helped the model adhere to syntactic conventions such as file headers and hunk boundaries. As we refined and adjusted these prompts—combining role conditioning, hidden CoT, and exemplar-based guidance—the model's performance improved, producing patches that were more accurate, more structurally consistent, and better aligned with the task intent.

4 Analysis and Results

4.1 Challenges with Pytests

Our initial evaluation strategy followed the methodology proposed in FEA-Bench, where model-generated patches are applied to source code and validated through associated unit tests. The goal was to measure functional correctness at two levels. First, whether the patches could be successfully applied, and second, whether they passed the unit tests. However, this approach encountered significant practical limitations when used with LLM-generated patches.

The main limitation was the frequent occurrence of formatting errors that made many patches inapplicable using git apply. Across 80 test cases (20 instances × 4 model configurations), roughly 88% of patches failed before reaching the execution phase. Common formatting issues included corrupt patch errors, where the diff structure was malformed due to missing or incorrect hunk headers (@@ -line,count +line,count @@), mismatched line counts, or extraneous markup tokens (e.g., Markdown artifacts). Context mismatch errors also caused failures when correctly formatted diffs did not match the actual file content, often due to whitespace, indentation, or comment differences.

Another major challenge involved context size. The model requires sufficient surrounding code to generate accurate patches, but providing entire codebases exceeds its context limits. As the input context increased, both Gemini and DeepSeek models performed worse in identifying the correct locations and edits. To mitigate this, we limited the context to problem descriptions, relevant code files, component signatures, and documentation generated by a separate documenter model. Despite these efforts, the generated context lines often still differed slightly

In our first iterations of patch generation, 100% of the patches were skipped during unit testing, indicating that none followed the correct diff format required for successful application. After refining the prompts to include explicit patch formatting instructions and incorporating few-shot examples of valid patches in both the user and system prompts, we observed a notable improvement. Out of 80 total patches, 10 were successfully applied, and 30% of those applied patches passed the unit tests. Tables 2 and 3 present the detailed breakdown across Gemini and DeepSeek models, as well as between the baseline and documenter/coder pipelines. Overall, the baseline models produced higher-quality patches, and Gemini outperformed DeepSeek in patch generation consistency. However, within the documenter/coder pipeline, only one patch, which was generated by DeepSeek, was successfully applied. Table 2 shows successful application out of 80 instances (patch was applied, result was either fail or pass).

Table 3 shows the pass results out of 80 instances (patch was applied and passed the unit test).

As a final experiment, we explored a hybrid pipeline that combines the strengths of both the baseline and documenter-coder approaches. The goal was to test whether giving the coder LLM both the structured oracle context from the baseline prompt (including problem statements, relevant code files, and component signatures) and the natural language design documentation from the documenter LLM could improve patch quality. The idea was that this setup would provide the coder model with both the raw technical context and a higher-level understanding of the required implementation. We tested four cross-model configurations, which are summarized in table 4. Each configuration received the full baseline prompt with oracle information as well as the design document produced by the respective documenter model. As in previous experiments, most patches could not be applied due to corrupt patch formatting, context mismatches, or missing headers. However, this setup slightly improved the apply rate, with 12 out of 80 patches successfully applied. This suggests that the additional documentation may help the model produce more structurally valid patches. Test passing rates, however, did not improve, only 2 out of 80 patches (2.5%) passed the unit tests.

After inspecting the majority of the patches that were not successfully applied, we found that many had the correct code logic but failed because of small formatting issues. These minor problems caused the patches to be rejected during unit testing. As a result, the unit-test pass rate was not a

Model Setting	Context Size	Input Used for Reasoning	Patch Generation
Baseline	Short	JSON task specification	DeepSeek prompt
Baseline	Short	JSON task specification	Gemini prompt
Documenter → Coder	Large	DeepSeek Documenter	DeepSeek Coder
Documenter → Coder	Large	Gemini Documenter	Gemini Coder
Documenter + Baseline → Coder	Large	DeepSeek Documenter	DeepSeek Coder
Documenter + Baseline → Coder	Large	Gemini Documenter	Gemini Coder
Documenter + Baseline → Coder	Large	Gemini Documenter	DeepSeek Coder
Documenter + Baseline → Coder	Large	DeepSeek Documenter	Gemini Coder

Table 1: Overview of model experiments

Model	Baseline	Documenter/Coder
Gemini	6	0
DeepSeek	3	1

Table 2: Successful application counts of pytests.

Model	Baseline	Documenter/Coder
Gemini	2	0
DeepSeek	1	0

Table 3: Successful pass result counts of pytests

reliable metric. A patch could be semantically correct, meaning it implemented the right logic, but still fail due to formatting mistakes. This made it hard to tell the difference between truly incorrect patches and correct ones with formatting problems. Additionally, from a practical point of view, a patch that finds the real issue and provides a valid fix, even without access to test cases, is still valuable for software engineers. Therefore, we added other ways to measure patch quality, including similarity scores to golden patches and an LLM-as-a-Judge evaluation.

Baseline + Documenter→ Coder	Application counts	Pass counts
DeepSeek → DeepSeek (DS→DS)	2	1
Gemini → Gemini (GM→GM)	3	0
DeepSeek → Gemini (DS→GM)	4	0
Gemini → DeepSeek (GM→DS)	3	1

Table 4: Patch pytest outcomes across four Baseline+ Documenter→Coder configurations.

4.2 Similarity Score

We developed a patch similarity scoring system that compares generated patches with golden, or ground-truth, patches without requiring the patches to actually apply. This approach checks whether the LLM understands what changes are needed, even if the formatting is not perfect. The similarity evaluation includes three metrics. The first is Line Similarity (50% weight), which uses Jaccard similarity on the set of added and removed code lines between the generated and golden patches, defined as Jaccard = |intersection| / |union|. This shows whether the same code changes are being made, regardless of formatting. The second metric is Sequence Similarity (30% weight), which uses Python's difflib.SequenceMatcher to measure the overall text similarity between normalized patches. This captures how similar the structure and writing of the changes are. The third metric is File Overlap (20% weight), which measures Jaccard similarity on the set of files modified by each patch to check if the model is targeting the correct files. The final similarity score is a weighted average of these three metrics: $S_{\text{weighted}} = 0.5S_{\text{line}} + 0.3S_{\text{seq}} + 0.2S_{\text{file}}$ Table 5 summarizes weighted average scores across the different model configurations.

As shown in Table 5, the baseline-oracle models generally performed better than the coder-documenter pipeline, suggesting that direct access to oracle information helps produce more accurate patches. We also observed that Gemini-based models showed more consistent results across tasks, which matches the trends seen in the unit test outcomes. High similarity scores (greater than 0.7) were often linked to cases where the file overlap was 1.0, meaning the model correctly identified which files needed changes.

In our hybrid baseline-coder configurations, the deepset-ai/haystack-7009 task using baseline-coder-DeepSeek-documenter-Gemini achieved a 0.996 weighted average similarity score, nearly identical to the golden patch. This same task also passed the unit tests, indicating that the similarity score can serve as a reliable measure of patch functionality. We also found that cross-model combinations showed promise—the mixed configurations (DeepSeek–Gemini and Gemini–DeepSeek) often matched or outperformed the same-model setups, suggesting potential benefits from model diversity.

4.3 LLM-as-a-Judge

While similarity score may provide a reference as to how related the generated patch is to the ground truth, we acknowledge that there is no singular way to write useful code. There are multiple methods and styles, restricting the utility of solely relying on similarity. We wanted to find another way to

Instance	Baseline-DS	Baseline-GM	Coder-DS / Doc-DS	Coder-GM / Doc-GM	Baseline Coder-DS / Doc-DS	Baseline Coder-GM / Doc-GM	Baseline Coder-DS / Doc-GM	Baseline Coder-GM / Doc-DS
haystack-7009	0.993	0.683	0.616	0.831	0.914	0.708	0.996	0.981
conan-12886	0.595	0.751	0.413	0.706	0.749	0.557	0.704	0.756
mmengine-609	0.665	0.685	0.184	0.707	0.496	0.634	0.504	0.537
pgmpy-1753	0.591	0.633	0.444	0.543	0.521	0.477	0.568	0.541
sagemaker-sdk-3432	0.077	0.581	0.348	0.338	0.633	0.564	0.536	0.727

Table 5: Similarity scores across instances and model configurations (DS = DeepSeek, GM = Gemini).

effectively evaluate patches.

Thus, we chose to use an LLM-as-a-Judge approach. In particular, given context about the repository and intended feature, LLMs were instructed to assign scores between 0-100 to generated prompts and documents. The LLMs were also prompted to consider the code quality and ability to solve the feature in their score. Full prompts can be found in Appendix A.

As part of this approach, we chose to reuse our LLMs from the experiment. In that way, we could also study whether an LLM might be biased towards its own code generations - contradicting other empirical measures.

Each pipeline as described in table 1 contained 20 individual tasks. Both Gemini and Deepseek assigned a score to every task (and scored documentation) resulting in 400 total scores assigned. The average was taken across each set of 20 tasks, and the results are shown in table 6. The highest scores are underlined.

Through the results, we found that Gemini preferred our pipeline feeding both the base context and the documentation produced by Deepseek to the Gemini coder had the best scoring patches overall. On the other hand, Deepseek scored the Gemini baseline as the highest. After taking an average over all scores produced by Gemini and Deepseek, the overall winner matched the one chosen by Gemini. These results could indicate that there is some validity to utilizing the 2-step pipeline.

Our results also allow us to provide a comparison as how different models perform in repo-level code generation tasks. As shown in the table, both Gemini and Deepseek scored the documentation produced by Gemini as better than the one produced by Deepseek. Additionally, we can analyze a visualization of the results in Figure 3 that solely focuses on pipelines involving a single model. If we compare each pair of experiments (baseline-gemini vs baseline-deepseek, coder-gemini-documenter-gemini vs coder-deepseek-documenter-deepseek, etc.), we can see that both Deepseek and Gemini

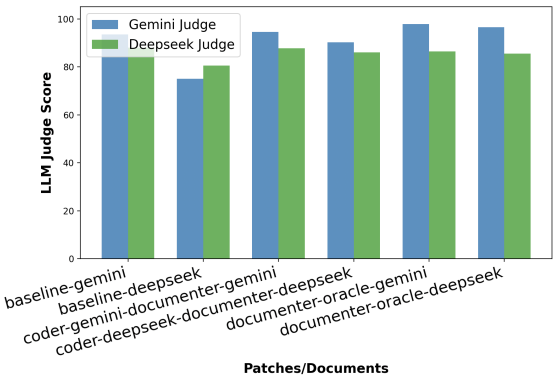

Figure 3: Average LLM scores across pipelines

tended to prefer the Gemini generations.

This outcome indicates that it is beneficial to use Gemini for these tasks. However, the best scoring pipeline used Deepseek documentation with Gemini code which creates a contradiction.

While performing this analysis, we must acknowledge the limitations of using LLM-as-a-Judge as the sole metric to rely on. It is possible that models may be biased for or against their own generations. In that way, we are not truly evaluating the quality of the patch and documentation.

5 Limitations

Our study has several limitations that should be considered when interpreting the results. First, although we used the Oracle-Lite dataset, which is the subset of the FEA-bench dataset to provide richer repository context, this variant still offers only a partial reconstruction of the full repository environment. Oracle-Lite includes additional files such as neighboring modules, configuration scripts (requirements.txt, setup.py), and documentation (README.md) but it does not capture the complete dependency graph, full test suite, or project-wide execution environment. Incorporating the full Oracle dataset or additional scraped repository content may provide a more faithful representation of real-world codebases.

Second, the FEA-Bench dataset introduces un-

Experiment	Gemini Avg	Deepseek Avg	Total Avg
Baseline-Gemini	93.50	88.00	90.750
Baseline-Deepseek	75.00	80.50	77.750
Documenter-Gemini-Coder-Gemini	94.55	87.75	91.150
Documenter-Deepseek-Coder-Deepseek	90.25	86.00	88.125
Baseline-Documenter-Gemini-Coder-Gemini	94.55	85.75	90.150
Baseline-Documenter-Deepseek-Coder-Deepseek	75.60	80.50	78.050
Baseline-Documenter-Deepseek-Coder-Gemini	97.75	87.25	92.500
Baseline-Documenter-Gemini-Coder-Deepseek	78.40	84.50	81.450
Documenter-Oracle-Gemini	97.90	86.50	92.200
Documenter-Oracle-Deepseek	96.60	85.50	91.050

Table 6: Average score assigned by LLM for each pipeline set up. For clarity, the role in the pipeline is followed by the generator model. (*Baseline-Documenter-Deepseek-Coder-Gemini = Pipeline giving both the base context and the documentation generated by Deepseek to the Gemini coder*)

avoidable heterogeneity because it is constructed from real-world GitHub repositories. Some repositories contain mixed-version environments, partial or outdated dependency specifications, or non-standard directory layouts. This variability can introduce noise into the LLM's input and makes it difficult to disentangle model errors from irregularities in the underlying repository structure. In certain cases, the extracted Oracle-Lite context does not map cleanly to a runnable environment, complicating patch evaluation and error attribution.

Finally, our two-stage pipeline depends heavily on the quality of the intermediate documentation produced by the first model. Because these summaries are not guaranteed to be complete or accurate, the pipeline may fail due to hallucinated or missing context, limiting the generalizability of our findings. Moreover, we did not systematically validate or quantify the fidelity of these intermediate representations, meaning that errors in the documentation step may have gone undetected. A more rigorous assessment of summary quality or enforcing structured, non–natural-language representations would help clarify whether failures originated from the documentation phase or the patch-generation phase

6 Future Works

Future research can build on these findings in several directions. A promising avenue is to pursue a more faithful reconstruction of repository environments, including deeper dependency resolution, extraction of full test suites, and incorporation of project-wide configuration and build files. Providing models with richer, better-structured input may reduce ambiguity and improve patch precision. A second direction is to investigate alternatives to natural-language intermediate representations. More structured formats such as symbolic dependency graphs, AST-based summaries, or function-level change plans may preserve critical task-local information while avoiding the brittleness of free-form explanations. Another important opportunity is to explore specialized code models and fine-tuning methods. Our study evaluated general-purpose LLMs (e.g., DeepSeek, Gemini), but recent advances in code-specific architectures suggest that models trained or adapted for repository-level reasoning may outperform general LLMs. Future work may examine (1) APIs designed specifically for code manipulation, (2) models trained with multi-file grounding objectives, and (3) lightweight fine-tuning or instruction-tuning strategies to improve patch fidelity and reduce hallucination in large-context settings. Such adaptations may help overcome the limitations observed when using off-the-shelf, general-purpose LLMs. Additionally, evaluating models that incorporate retrieval-augmented code understanding or long-context attention mechanisms could offer insights into how architectural improvements affect repository-scale reasoning. Finally, further experimentation with diff-based evaluation metrics can provide a more granular perspective on model behavior. Understanding how small syntactic or structural deviations propagate into functional failures may guide the design of more robust patch-generation frameworks.

7 Conclusion

Our experiments reveal several structural limitations in applying a two-stage abstraction pipeline

to repository-level code repair. Although intuitively appealing, the use of an intermediate documentation-generating model is not automatically beneficial. This stage demands exceptionally reliable summaries, because any omission, misinterpretation, or hallucinated detail becomes amplified when the second model attempts to generate a patch. Rather than clarifying the task, the abstraction layer often introduces subtle inaccuracies that distort the true intent of the required change.

Moreover, our findings show that high-fidelity, task-local signals are more important than human-readable summaries. Precise code edits rely on exact variable names, control-flow constraints, import paths, and file-level interactions, details that natural-language descriptions tend to soften, paraphrase, or omit. As a result, models conditioned on summaries frequently produce syntactically plausible but semantically incorrect patches. This aligns with a broader pattern: natural-language abstraction can degrade execution accuracy when the task requires fine-grained, line-level reasoning.

We also observe that longer context does not guarantee better context. Even though the Oracle-Lite dataset provides more extensive repository information, simply expanding the context window can overwhelm the model. Additional files and documentation often introduce irrelevant or conflicting signals, increasing the cognitive load and distracting the model from the specific modification required. In several cases, longer context even degraded performance by prompting the model to follow false or unrelated instructions.

Together, these challenges underscore that repository-level code repair remains fundamentally unsolved. Current LLMs still struggle to maintain global reasoning across multi-file interactions, dependency structures, and long-range contextual cues. They are highly sensitive to exact syntax, indentation, and interface contracts, making patch generation brittle and error-prone. Even minor deviations that appear harmless to a human developer can result in non-functional patches.

Despite these challenges, our research provides meaningful insights. By systematically comparing raw long-context inputs with processed, human-readable summaries, we shed light on which forms of context help LLMs and which forms harm them. Our negative results are informative: they highlight that naïve abstraction pipelines may not improve performance and can even worsen precision at execution time. These insights help clarify where future research should focus such as developing more faithful intermediate representations, improving retrieval and grounding methods, or designing specialized models capable of robust multi-file reasoning. In this way, the study advances understanding of the limitations of current LLMs and defines clearer pathways for improving repository-level code repair systems.

8 References

References

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues? *Preprint*, arXiv:2310.06770.

Haiyang Li. 2025. Mrg-bench: Evaluating and exploring the requirements of context for repository-level code generation. *Preprint*, arXiv:2508.02998.

Wei Li, Xin Zhang, Zhongxin Guo, Shaoguang Mao, Wen Luo, Guangyue Peng, Yangyu Huang, Houfeng Wang, and Scarlett Li. 2025. FEA-bench: A benchmark for evaluating repository-level code generation for feature implementation. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 17160–17176, Vienna, Austria. Association for Computational Linguistics.

Dianshu Liao, Shidong Pan, Xiaoyu Sun, Xiaoxue Ren, Qing Huang, Zhenchang Xing, Huan Jin, and Qinying Li. 2024. $A^3 - codgen : Arepository - levelcodegenerationframeworkforcodereusewithlocal - aware, global - aware, andthird - party - library - aware$. *Preprint*, arXiv:2312.05772.

Stefano Rando, Luca Romani, Alessio Sampieri, Luca Franco, John Yang, Yuta Kyuragi, Fabio Galasso, and Tatsunori Hashimoto. 2025. Longcodebench: Evaluating coding llms at 1m context windows. *Preprint*, arXiv:2505.07897.

A Prompts

A.1 Baseline Prompts

"""You are an expert software engineer implementing bug fixes and features in mature codebases.

You will be given a problem description and repository context. Your goal is to produce a CORRECT, MINIMAL git diff patch that implements the requested behavior and would pass the relevant unit tests.

Before writing the patch, you must internally (without printing the reasoning): 1. Infer the expected behavior from the description. 2. Identify

the exact files and functions/classes where the behavior should be implemented. 3. Decide the correct abstraction level for the change (local helper vs central dispatcher/router/base class) so ALL relevant code paths exhibit the new behavior. 4. Plan a small, targeted patch that follows the existing coding style and patterns.

Then output ONLY the final patch, in unified diff format (git diff style), that can be applied with ‘git apply‘.

Your patch must: 1. Be in unified diff format (git diff style) that can be applied with ‘git apply‘ 2. Include all necessary changes to implement the feature 3. Follow the coding style of the existing codebase 4. Be syntactically correct and complete 5. NOT include ’index <hash>..<hash>’ lines 6. NOT use placeholder hashes like ’1234567’ or ’abcdefg’ - these will cause the patch to fail 7. End the patch with a blank line 8. NOT add any unit tests. They will be provided 9. Start with: diff –git a/<path> b/<path> Then: — a/<path> Then: +++ b/<path> Then one or more hunks starting with: @@ -<old_start>,<old_len> +<new_start>,<new_len> @@ Never output a line starting with ’@@’ unless it is preceded (above) by the correct ’diff –git’, ’—’, and ’+++’ lines for that same file.

IMPORTANT: Each file diff should follow this exact format: diff –git a/path/to/file.py b/path/to/file.py — a/path/to/file.py +++ b/path/to/file.py @@ -line,count +line,count @@ context context lines -removed lines +added lines

Output ONLY the git diff patch, starting with "diff –git" lines. Do not provide any explanations - only relevant code."""

A.2 Documenter Prompts

"""You are an expert software engineer tasked with creating documents that other software engineers can use to implement new features.

Given a problem description and repository context, generate a detailed, specific document explaining how to implement the requested feature. Think about what features would be useful for you to have if you were going to code the feature.

Your document should: 1. Be as specific possible without any ambiguity. Another software engineer should be able to clearly follow the steps outlined in your document to implement the feature 2. Synthesize a large amount of context while retaining as much valuable information as possible 3. Make clear to another engineer how to follow the coding style of the existing codebase 4. You may provide a repository overview, a clear implementation plan, pseudo-code in relevant files, or any other information that you believe would be helpful 5. Your call to action to the other software engineer should be to generate a git diff patch that implements the requested feature. 6. DO NOT ask for any other information (like adding unit tests) """

A.3 Coder Prompts

"""You are an expert software engineer implementing bug fixes and features in mature codebases.

You will be given a a design document. Your goal is to produce a CORRECT, MINIMAL git diff patch that implements the requested behavior and would pass the relevant unit tests.

Before writing the patch, you must (internally, without printing the reasoning): 1. Infer the expected behavior from the description. 2. Identify the exact files and functions/classes where the behavior should be implemented. 3. Decide the correct abstraction level for the change (local helper vs central dispatcher/router/base class) so ALL relevant code paths exhibit the new behavior. 4. Plan a small, targeted patch that follows the existing coding style and patterns.

Then output ONLY the final patch, in unified diff format (git diff style), that can be applied with ‘git apply‘.

Your patch must: 1. Be in unified diff format (git diff style) that can be applied with ‘git apply‘ 2. Include all necessary changes to implement the feature 3. Follow the coding style of the existing codebase 4. Be syntactically correct and complete 5. NOT include ’index <hash>..<hash>’ lines 6. NOT use placeholder hashes like ’1234567’ or ’abcdefg’ - these will cause the patch to fail 7. End the patch with a blank line 8. NOT add any unit tests. They will be provided 9. Start with: diff –git a/<path> b/<path> Then: — a/<path> Then: +++ b/<path> Then one or more hunks starting with: @@ -<old_start>,<old_len> +<new_start>,<new_len> @@ Never output a line starting with ’@@’ unless it is preceded (above) by the correct ’diff –git’, ’—’, and ’+++’ lines for that same file.

IMPORTANT: Each file diff should follow this exact format: diff –git a/path/to/file.py b/path/to/file.py — a/path/to/file.py +++ b/path/to/file.py @@ -line,count +line,count

@@ context context lines -removed lines +added lines

Output ONLY the git diff patch, starting with "diff –git" lines. Do not provide any explanations - only relevant code."""

A.4 Baseline-Coder Prompts

"""You are an expert software engineer implementing bug fixes and features in mature codebases.

You will be given: 1. Repository information including problem descriptions, code files, and component signatures. 2. A design document generated by another model that details how to implement the feature.

Your goal is to produce a CORRECT, MINIMAL git diff patch that implements the requested behavior and would pass the relevant unit tests.

Before writing the patch, you must (internally, without printing the reasoning): 1. Infer the expected behavior from the description. 2. Identify the exact files and functions/classes where the behavior should be implemented. 3. Decide the correct abstraction level for the change (local helper vs central dispatcher/router/base class) so ALL relevant code paths exhibit the new behavior. 4. Plan a small, targeted patch that follows the existing coding style and patterns.

Then output ONLY the final patch, in unified diff format (git diff style), that can be applied with ‘git apply‘.

Your patch must: 1. Be in unified diff format (git diff style) that can be applied with ‘git apply‘ 2. Include all necessary changes to implement the feature 3. Follow the coding style of the existing codebase 4. Be syntactically correct and complete 5. NOT include ’index <hash>..<hash>’ lines 6. NOT use placeholder hashes like ’1234567’ or ’abcdefg’ - these will cause the patch to fail 7. End the patch with a blank line 8. NOT add any unit tests. They will be provided 9. Start with: diff –git a/<path> b/<path> Then: — a/<path> Then: +++ b/<path> Then one or more hunks starting with `@@ -<old_start>,<old_len> +<new_start>,<new_len> @@`. Never output a line starting with `"@@"` unless it is preceded (above) by the correct `’diff -git’`, `’−’`, and `’+++’` lines for that same file.

IMPORTANT: Each file diff should follow this exact format: diff –git a/path/to/file.py b/path/to/file.py — a/path/to/file.py +++ b/path/to/file.py @@ -line,count +line,count @@ context context lines -removed lines +added lines

Output ONLY the git diff patch, starting with "diff –git" lines. Do not provide any explanations - only relevant code."""

A.5 LLM as a Judge Prompts

PATCH_SYSTEM_INSTRUCTION = """You will be given a feature description and a git diff patch. You job is to score this patch on a scale of 1-100. Please consider code quality and the patch’s ability to implement the feature in your score. Make sure to output the score alone without any reasoning. """

DOCUMENT_SYSTEM_INSTRUCTION = """You will be given a feature description and implementation documentation. You job is to score the documentation on a scale of 1-100. Please consider documentation quality and how well another software engineer could follow it to code the feature. Make sure to output the score alone without any reasoning. """