Hierarchical Knowledge Injection for Improving LLM-based Program Repair

Ramtin Ehsani
Drexel University
Philadelphia, PA, USA
ramtin.ehsani@drexel.edu

Esteban Parra
Belmont University
Nashville, TN, USA
esteban.parrarodriguez@belmont.edu

Sonia Haiduc
Florida State University
Tallahassee, FL, USA
shaiduc@fsu.edu

Preetha Chatterjee
Drexel University
Philadelphia, PA, USA
preetha.chatterjee@drexel.edu

***Abstract*—Prompting LLMs with bug-related context (e.g., error messages, stack traces) improves automated program repair, but many bugs still remain unresolved. In real-world projects, developers often rely on broader repository and project-level context beyond the local code to resolve such bugs. In this paper, we investigate how automatically extracting and providing such knowledge can improve LLM-based program repair. We propose a layered knowledge injection framework that incrementally augments LLMs with structured context. It starts with the *Bug Knowledge Layer*, which includes information such as the buggy function and failing tests; expands to the *Repository Knowledge Layer*, which adds structural dependencies, related files, and commit history; and finally injects the *Project Knowledge Layer*, which incorporates relevant details from documentation and previously fixed bugs. We evaluate this framework on a dataset of 314 bugs from BugsInPy using two LLMs (Llama 3.3 and GPT-4o-mini), and analyze fix rates across six bug types. By progressively injecting knowledge across layers, our approach achieves a fix rate of 79% (250/314) using Llama 3.3, a significant improvement of 23% over previous work. All bug types show improvement with the addition of repository-level context, while only a subset benefit further from project-level knowledge, highlighting that different bug types require different levels of contextual information for effective repair. We also analyze the remaining unresolved bugs and find that more complex and structurally isolated bugs, such as *Program Anomaly* and *GUI* bugs, remain difficult even after injecting all available information. Our results show that layered context injection improves program repair and suggest the need for interactive and adaptive APR systems.**

***Index Terms*—automated program repair, large language models, knowledge injection, in-context learning**

I. Introduction

Large Language Models (LLMs) have opened new possibilities in Automated Program Repair (APR), offering the ability to generate patches with minimal supervision. Despite this promise, LLMs often struggle to grasp the deeper context behind bugs, leading to unreliable or non-applicable fixes [1]–[3]. A key limitation is that many generated patches lack awareness of project-level constraints, dependencies, or intent [4], [5]. Recent findings show that the performance of LLMs drops by more than 50% when bug fixes require information beyond the buggy function [6]. Building effective LLM-based tools for APR requires a better understanding of when and why these models succeed or fail, an area that is still underexplored.

Research has shown that incorporating additional relevant information in prompts can enhance the performance of LLMs in program repair. Zhao et al. [7] extract design rationales from issue threads, but the approach relies heavily on developer discussions, which are often noisy or inconsistent [8]. Parasaram et al. [9] observed improved LLM performance in APR by incorporating *bug-related facts*, such as error messages, stack traces, and buggy code snippets. HAFix explored commit history to provide historical context for buggy functions, showing improvements on a small dataset [10]. While these approaches highlight the value of adding relevant context, most are limited to the immediate surroundings of the buggy code. It is known that bugs often span multiple files and components [11]. Thus, including a broader context, such as related files, dependencies, and documentation, could produce more accurate patches. However, Parasaram et al. [9] showed that there is no universal set of information that works across all scenarios and bugs. Their tool uses a Random Forest model to select types of contextual information to include in the prompt based on features such as prompt length, repository ID, and code complexity. However, their model lacks interpretability and ignores bug type. Different bug types require different contextual cues to be solved [12], and treating them uniformly could lead to producing ineffective or suboptimal repairs.

In this paper, we investigate how the injection of *repository* and *project*-specific contextual information on top of information surrounding buggy code affects the ability of LLMs to repair different types of bugs. We use a subset of the BugsInPy dataset [13], consisting of 314 bug–patch pairs from Python open-source repositories on GitHub. Using an existing taxonomy [12], we manually categorized each bug into one of six types, including *Program Anomaly*, *GUI*, *Network*, etc. We organize the contextual information into three layers: *Bug Knowledge*, *Repository Knowledge*, and *Project Knowledge*. By progressively injecting knowledge from each layer, we aim to understand which kinds of context can help fix specific types of bugs. We evaluate our approach using two LLMs (Llama 3.3 and GPT-4o-mini), demonstrating that improvements from layered knowledge injection are consistent across models with different architectures and parameter sizes. Additionally, we perform a detailed error analysis to uncover why LLMs struggle with specific types of bugs, even after providing contextual information. Our research questions are:

RQ1: To what extent can LLMs resolve different types of bugs using only immediate bug knowledge? We inject bug-related context, adopted from Parasaram et al. [9], as

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00122

the *Bug Knowledge Layer* to assess LLMs' performance in bug repair for different types of bugs. Our best-performing model fixes 207/314 bugs (65%), surpassing the previously reported fix rate of 56% [9]. This improved performance could be attributed to structured in-context knowledge injection as well as using newer versions of LLMs (Llama3.3 vs. Llama3). However, our fix rate leaves room for improvement, indicating that local context alone is often insufficient, especially for complex bug types like *Program Anomaly* and *Network*, which frequently remain unresolved in this layer.

RQ2: How does injecting repository-level knowledge enhance LLM-based repair of different bug types? We re-patch all unresolved bugs from RQ1 after adding additional context from the *Repository Knowledge Layer*, including co-occurring files, structural dependencies, and commit history. This leads to a significant improvement of 9% for our best model, fixing 28 more bugs and reaching a 74% fix rate (235/314). All bug types benefit from this layer, showing that repository-level context is crucial for resolving bugs that local information alone cannot address.

RQ3: How does injecting project-level knowledge further enhance LLM-based repair of different bug types? We further augment the information available for unresolved bugs from RQ2 with additional context from the *Project Knowledge Layer*, including documentation and previously resolved issues. This results in our best model fixing 15 more bugs, reaching a final fix rate of 79% overall (250/314), a significant improvement of 23% over previous work [9]. Improvements in this layer are limited to *Program Anomaly*, *GUI*, and *Network* bugs, suggesting that project knowledge offers benefits for specific bug types.

RQ4: Which bug types remain challenging for LLMs, even after injecting all layers of contextual knowledge? We analyze the remaining unresolved bugs after all context layers are applied. We found that these bugs often require reasoning about user-facing behavior or implicit runtime conditions and feedback. These bugs also exhibit higher complexity, suggesting that current LLMs still struggle with structurally isolated or complex repair tasks.

The key contributions in this paper are:

- A layered knowledge injection framework for LLM-based APR, structuring contextual information into three hierarchical levels: *Bug*, *Repository*, and *Project* knowledge.
- Evaluation against two baselines: (1) a prior fine-grained fact selection approach by Parasaram et al. [9], and (2) an "all-at-once" baseline where all contextual information is injected simultaneously.
- A bug-type-specific analysis showing how different context layers impact the repair of different types of bugs.
- An error analysis (e.g., bug types, complexity) of unresolved bugs, offering insights into the limitations of current LLMs and outlining concrete directions for future work.
- Replication package[1] with our bug-type annotations, prompt designs, and evaluation pipeline to support reproducibility and future work on context-aware APR with LLMs.

[1]**https://github.com/SOAR-Lab/llm-apr-knowledge-injection**

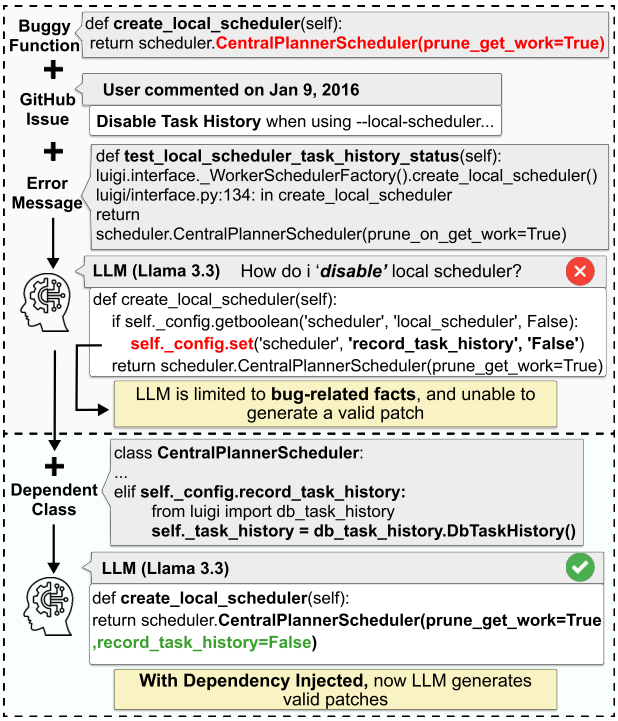

Fig. 1. Comparison of Patch Generation with Different Information

II. Motivating Example

Prior work has shown that incorporating contextual information (e.g., stack traces) in prompts enhances the bug-fixing performance of LLMs [5]. Building on this work, Parasaram et al. [9] introduced a set of *bug-related facts* to provide additional context for LLMs. These facts include: *Buggy Function*, *Failing Tests*, *Error Information*, *Runtime Information*, *Angelic Forest*, *Buggy Class Declaration*, *Method Signatures*, and *GitHub Description*. Including such facts significantly improved LLM-generated fixes compared to prompting with no additional context. However, their approach achieved a maximum fix rate of 56% (177/314 bugs in BugsInPy). This raises the question: What about the remaining 137 bugs? What prevented the LLM from fixing them despite the addition of varied *bug-related contextual information*?

We hypothesize that the key missing piece is the broader *repository and project-level understanding*. In previous work, contextual signals are limited to the buggy function and its immediate surroundings, thus overlooking rich sources of information embedded in code structure, version history, and documentation. These are resources that developers rely on when fixing real-world bugs [14]–[16].

Consider the example in Figure 1, a bug in the *luigi* project (link). This bug, located in the *create_local_scheduler* function, can be fixed by disabling task history. While the fix may appear trivial to a developer familiar with the codebase, Llama 3.3 fails to generate the correct patch when prompted only with the *bug-related facts*. It cannot determine which

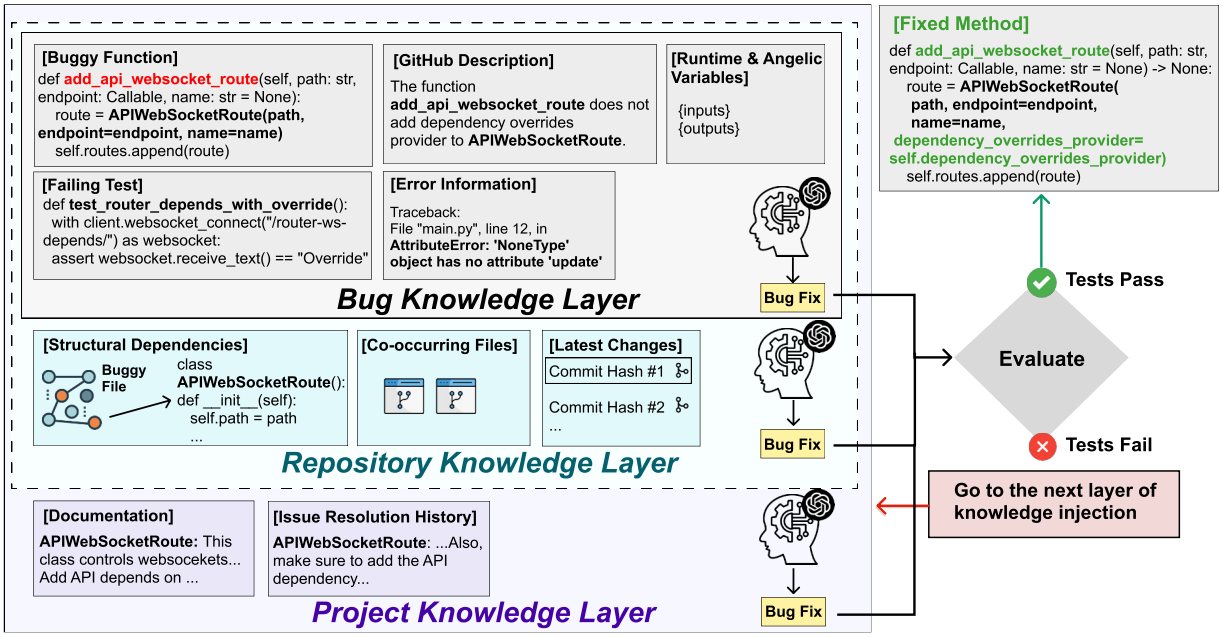

Fig. 2. Layered Knowledge Injection for Automated Program Repair with LLMs

class attribute to modify and instead hallucinates the existence of a variable *self._config*. However, once dependent class information *CentralPlannerScheduler* (extracted from related files) is injected into the prompt, the model produces the correct fix by disabling the *record_task_history* attribute. This shows how structural context improves the model's understanding. More importantly, this example showcases that bugs do not exist in isolation, and solving many of them requires knowledge that extends beyond the local scope, spanning related files, structural dependencies, and even discussions buried in documentation or past issue threads. As we show in later sections, this is particularly true for complex bug types such as *GUI*, *Program Anomaly*, and *Configuration* bugs.

Overall, the landscape of LLM-guided APR is fast evolving. With the advent of large-context LLMs (e.g., 128k token window in GPT-4o-mini), the challenge has shifted from fitting a small subset of facts into limited prompts to *understanding what information is useful and how to strategically provide it as model input to guide effective repair.* Recent research also suggests that *knowledge injection* (providing relevant in-context learning at inference time) can be more effective and generalizable than fine-tuning [17]. However, providing too much information or unfiltered context may not help and can even be detrimental in APR [9]. Therefore, we need structured ways to select, automatically extract, and prioritize the most relevant information from vast and noisy software ecosystems.

III. Methodology

We propose a *layered knowledge injection pipeline* for LLM-based bug repair. Figure 2 presents an overview of our approach. We incrementally inject contextual information through three distinct layers. (1) **Bug Knowledge Layer**: We begin by injecting local bug-related information (e.g., buggy function, error message, GitHub issue description). This lightweight context is often sufficient for fixing straightforward bugs. (2) **Repository Knowledge Layer**: For bugs that remain unresolved after layer (1), we proceed to inject repository-level context, including related files (co-commits), structural dependencies, and recent commit history. This helps the model understand a broader context beyond localized code snippets and error information. (3) **Project Knowledge Layer**: For the bugs still unresolved even after layer (2), we further inject knowledge extracted from documentation and previously resolved bugs. These sources provide project-level insights, such as intended system behavior, fix patterns, and edge cases.

This layered approach offers several advantages. First, it allows simpler bugs to be fixed with minimal input, conserving tokens and computation. Second, it scales context progressively, injecting more information only when necessary. Third, it enables analysis of which bug types benefit from specific contextual signals, offering insights into LLM capabilities across different bug categories and levels of complexity.

A. Dataset

We use the dataset of Parasaram et al. [9], which includes 314 instances from the BugsInPy benchmark [13]. Each instance includes the link to the fix commit that resolves the bug and the link to the buggy commit representing the repository state before the fix. This data contains bugs fixed by changes to a single function. Since our focus is on generating patches for APR, for bug localization we use *Function-granular Perfect Fault Localization*, as recommended in prior work [9], [18]. However, LLMs will still have to identify the faulty lines within a buggy function.

We manually annotate the dataset using the bug categorization taxonomy proposed by Catolino et al. [12]. This taxonomy

TABLE I
DESCRIPTIONS, EXAMPLES, AND FREQUENCY OF BUG TYPES IN OUR DATASET

Bug	Description	Example	Count
Program Anomaly	Concerned with specific circumstances such as exceptions, problems with return values, and unexpected crashes due to issues in the logic (rather than, e.g., the interface) of the program	"Program terminates prematurely before all execution events are loaded in the model"	187
Network	Bugs having as root cause connection or server issues, due to network problems, unexpected server shutdowns, or communication protocols that are not properly used within the source code	"During a recent reorganization of code a couple of weeks ago, SSL recording no longer works"	47
Configuration	Bugs concerned with building configuration files. Most of them are related to problems caused by (i) external libraries that should be updated or fixed and (ii) wrong directories, file paths, or patterns in XML or manifest artifacts	"JEE5 Web model does not update on changes in web.xml"	34
GUI-Related	Bugs occurring in the Graphical User Interface of a software. It includes issues referring to (i) stylistic errors, i.e., screen layouts, element colors and padding, text box appearance, and buttons, as well as (ii) unexpected failures appearing to the users in the form of unusual error messages	"Text when typing in input box is not viewable."	28
Performance	Bugs reporting performance issues like memory overuse, energy leaks, and methods causing endless loops	"Loading a large script in the Rhino debugger results in an endless loop (100% CPU utilization)"	16
Permission/ Deprecation	Bugs related to two main causes: (i) they are due to the presence, modification, or removal of deprecated method calls or APIs; (ii) problems related to unused API permissions are included	"setTrackModification(boolean) not deprecated; but does not work"	2

includes 9 categories: *Configuration Issue*, *Network Issue*, *Database-Related Issue*, *GUI-Related Issue*, *Performance Issue*, *Permission/Deprecation Issue*, *Security Issue*, *Program Anomaly Issue*, and *Test Code-Related Issue*. We use this taxonomy because it is comprehensive, capturing common root causes across diverse, real-world bugs (1,280 bug reports from 119 popular projects). The annotation instructions, including the definitions of each bug type and examples, are adapted from Catolino et al. and included in our replication package.

Two graduate students (each with 3+ years of experience in programming and qualitative research) conducted the annotation process. For each data point, we examined both textual and code-related data, including the *GitHub issue thread*, *pull request*, and the *fix commit*. We followed an iterative analysis comprising multiple sessions. To ensure annotation reliability and mitigate bias, we employed Cohen's Kappa inter-rater agreement [19]. In the first round, both annotators independently labeled an initial set of 50 data points, which yielded a Kappa score of 0.45. Follow-up discussions were conducted to refine the taxonomy's interpretation and resolve all disagreements. In the second round, an additional 50 data points were annotated, which yielded a Kappa score of 0.8, indicating a high level of agreement [19]. With this established consistency, the remaining 214 data points were divided between the two annotators and labeled independently.

Table I presents the descriptions, examples, and distribution of bug types in our dataset. Six of the nine categories are represented; we did not find any *Database-Related*, *Security*, or *Test Code-Related* bugs. Among the represented bug types, *Program Anomaly* is the most frequent, while *Permission/Deprecation* is the least. Our dataset spans 16 well-known Python projects (e.g., *pandas, youtube-dl, scrapy)*, each with high star counts (ranging from 16k to 135k), contributor counts (25 to 463), and open issues (29 to 4172). Full project-level statistics, including stars, contributors, open issues, and bug distributions, are provided in our replication package.

B. Prompting Strategy

We use Chain-of-Thought prompting, a proven strategy for guiding LLMs to process context step by step, as demonstrated in APR and other tasks [9], [20]–[22]. We adopt a prompt structure similar to Parasaram et al. [9], with a few key adjustments. We enforce standardized outputs by requiring all code to be wrapped in triple backticks. We also omit textual explanations to ensure the output contains only code. We tested our approach on 10 bug instances using GPT-4o-mini and Llama 3.3, running each bug 10 times per model. We achieved consistent outputs across all 200 runs. Due to space constraints, we include the detailed prompt formats for each knowledge layer in our replication package.

C. Research Questions and Information Types in Each Layer

RQ1: To what extent can LLMs resolve different types of bugs using only immediate bug knowledge?

Immediate bug knowledge is the contextual information directly related to the buggy code, including: *Buggy Function*, *Failing Tests*, *Error Information*, *Runtime Variables*, *Angelic Variables*, and *GitHub Description*. We incorporate them in the first layer of the knowledge injection pipeline, referred to as the **Bug Knowledge Layer**.

The information types in this layer are adopted from Parasaram et al. [9], and serve as a **baseline** for evaluating the impact of injecting other contextual information in subsequent layers (RQ2 and RQ3).

Buggy Function. This is the buggy code the LLM should fix. For each data instance, the function requiring repair is extracted using the fix commit (patch) provided by developers.

Failing Tests. Each bug instance also contains passing and failing tests. The buggy code can cause one or more tests to fail. Providing these failed tests could help the LLM understand the possible cause of the bug [23].

Error Information. We supply LLMs with error messages and stack traces generated by failing tests, as these can help models identify buggy code [23], [24]. To obtain this information, we first cloned each project from GitHub and reverted the repository to its state before the fix commit. We then used Anaconda to install the required dependencies and ran the failing tests to capture the corresponding error outputs.

Runtime and **Angelic Variables.** We extract two types of dynamic information: 1) *Runtime information*: values and types of function parameters and local variables during failing test execution; 2) *Angelic forest*: the expected values that, if assigned to those variables during execution, would cause the failing test to pass. This behavioral data can improve LLM

performance in bug fixing [9]. Prior work showed that such information supports synthesis-based program repair [25]. The values are extracted from correct program versions using instrumentation. Since runtime and expected values can be lengthy, especially when multiple test cases are available, we limit each bug instance to at most three input-output cases for both runtime and angelic variables.

GitHub Description. Providing textual descriptions of a bug can improve LLM-generated fixes [26], [27]. We extract the title and body description of issue reports associated with each bug in our dataset. We observed that 69 of 314 instances in the dataset do not have a linked report. To address this, we tried to extract the corresponding pull request. For 24 data points, no corresponding issue or pull request was found. We manually reviewed all linked issues and pull requests to ensure they only describe the bug and do not reveal the actual fix.

RQ2: How does injecting repository-level knowledge enhance LLM-based repair of different bug types?

In the **Repository Knowledge Layer**, we go beyond the immediate bug context by incorporating additional information from the repository: *the buggy file, related files, and prior fix history*. Each type of information is retrieved from the state of the GitHub repository before the fix. We apply this knowledge injection to the bugs unresolved after the *Bug Knowledge Layer*, allowing us to assess the impact of repository-level context on repair outcomes.

Co-occurring Files. Prior APR research has shown that leveraging historical context can improve repair outcomes [4], [28]. To identify files related to the one containing the buggy function, we analyze past commits and extract those that have most frequently been committed alongside the buggy file [29]. We refer to these as *Co-occurring Files*. These files suggest a strong logical or functional connection, indicating that they often work together or depend on one another [30]. Co-occurring files are not directly used in prompts. We use them as a heuristic to identify where the buggy function may be invoked elsewhere in the codebase, aiding in the discovery of structural dependencies discussed next.

Structural Dependencies. Understanding the context of the buggy function is critical for generating effective patches [1], [31], [32]. To capture structural dependencies, we examine both directions: the functions and classes invoked by the buggy function and those that invoke the buggy function elsewhere in the codebase. For each buggy function, we use Python's AST and Jedi [33] library to parse its body and extract referenced functions and classes. We then use the file's import statements to resolve where those declarations originate and extract their definitions from the relevant files. In the other direction, we identify functions and classes that invoke the buggy function. Since usages can be widespread and introduce noise especially for utility functions, we limit this search to the set of previously identified *Co-occurring Files*. We then apply the same dependency resolution process to extract only the most relevant calling contexts.

Latest Changes. Including historical commit information related to the buggy function can improve the bug-fixing performance of LLMs [34]. Therefore, by analyzing the Git history of the repository, we extract the most recent commit made to the buggy function before the fix. This information suggests potential sources of error in the buggy function.

RQ3: How does injecting project-level knowledge further enhance LLM-based repair of different bug types?

In the **Project Knowledge Layer**, we inject *project documentation and developer knowledge from past resolved issues*. We inject them to the unresolved bugs after the *Repository Knowledge Layer*. The goal is to give the LLM a holistic view of the project, its constraints, and past bug-fixing strategies.

Documentation. Documentation can help LLMs infer code intent [35], [36]. We extract information about buggy functions' usage, limitations, and behavior from the project's official documentation. Since documentation can be extensive, we isolate the most relevant parts by applying a dense retrieval pipeline [37], [38]. First, we split the documentation into equal-sized chunks and embed them using Sentence Transformer embeddings. We store the embeddings in a FAISS (Facebook AI Similarity Search) [39] vector index, enabling efficient similarity search against the buggy function. We use the full body of each buggy function, including the APIs it calls, as a query to retrieve the most relevant documentation chunks based on semantic similarity. We then use LangChain's QA Retrieval Agent [40], to query the embedded documents and extract four key pieces of information: **(1)** What is the correct behavior of the *buggy_function*? **(2)** What is the intended usage of the *buggy_function* and the APIs it uses? **(3)** What are the constraints, edge cases, or usage patterns for APIs used in *buggy_function*? **(4)** What are the limitations, warnings, or best practices of using APIs in the *buggy_function*?

Issue Resolution History. We focus on patched issues that occurred before the fix date of the target bug. These past issues often contain relevant knowledge about similar bugs and their resolutions [41], [42]. To identify the most relevant issues, we apply dense retrieval to compare each past issue to the buggy function. We consider both the code (patches) and the natural language (titles, descriptions, discussions, and labels). Inspired by previous studies [37], [38], we compute a similarity score for each dimension, textual and code, and combine them using an equal weighted average (50% text, 50% code). The issues, along with their metadata, are stored in a FAISS [39] vector index for efficient retrieval. We then use LangChain's QA Retrieval Agent, powered by GPT-4o-mini, to extract the following key insights: **(1)** What relevant API behaviors, bugs, or patterns discussed in the past could inform fixing the *buggy_function*? **(2)** What concrete fix strategies were applied in similar scenarios to those in the *buggy_function*? **(3)** What specific code changes from past fixes could guide repairing the buggy function *buggy_function*? **(4)** What key developer insights, recommendations, or concerns discussed in past fixes are relevant to resolving the *buggy_function*?

RQ4: Which bug types remain challenging for LLMs, even after injecting all layers of contextual knowledge?

We conduct an error analysis of the bugs that are unresolved

after injecting all three layers of contextual knowledge.

First, for the unresolved bugs, we manually examine whether the five types of repository and project-level information were available and usable, since in real-world settings, such context may not always exist or be extractable.

Second, our goal is to identify the types and characteristics of the bugs that remain challenging for LLMs. To examine whether unresolved bugs are more complex than successfully fixed ones, we compute the following code complexity metrics using Python's widely-used Radon package [43]–[45]: a) **Cyclomatic Complexity**, which measures control flow complexity based on decision points; b) **Maintainability Index**, a composite metric reflecting how easily the code can be understood and maintained; c) **Lines of Code (LOC)**, capturing function length; and d) **Halstead Metrics** (Volume, Difficulty, and Effort), which reflect the cognitive complexity based on operators and operands. As we are comparing two groups (resolved vs. unresolved bugs) on various continuous metrics, we run statistical tests to assess the significance of differences. Specifically, as the data does not follow a normal distribution (verified using the Shapiro–Wilk test), we use the non-parametric Mann–Whitney U test [46]. Additionally, we compute Cohen's d [47] to measure the effect size and indicate which group tends to exhibit higher complexity across each metric. This analysis helps us assess whether unresolved bugs may require more sophisticated reasoning or deeper understanding, potentially posing difficulties to current LLMs.

D. Baselines

We compare our approach against two baselines: (1) Parasaram et al. [9]'s approach, which leverages various bug-related contextual information to produce patches; and (2) an approach where information from all three layers (bug, repository, and project knowledge) is injected simultaneously for every bug. The second baseline allows us to assess whether progressive, hierarchical injection is more effective than providing all context at once.

E. Evaluation Metrics

The nondeterminism of LLMs can lead to varying outcomes across responses, which presents challenges when analyzing results. To address this, we evaluate the generated responses using the *pass@k* measure, which represents the probability that at least one out of k queries successfully resolves the problem.

$$\text{Pass@k} = \mathbb{E}_Q \left[1 - \frac{\binom{n-C}{k}}{\binom{n}{k}} \right] \tag{1}$$

Here, $\mathbb{E}_Q$ denotes the expectation over the set of LLM responses for the set of queries (prompts) Q, n is the total number of responses obtained from the LLM where $n > k$, and C is the number of successful responses among them. For our program repair task, we consider a response successful if the extracted patch satisfies a correctness criterion, which we approximate by passing the test suite. We choose $n = 10$ and report Pass@k with k=1, 3, 5, following prior studies [9], [10].

We also report the number of fixed bugs in our dataset (*#fixed*). This is a common metric for evaluating APR tools that iteratively generate and test patches, indicating the number of bugs for which at least one generated patch passes all tests [9], [48]. The #fixed is measured by:

$$\#\text{fixed } (\text{LLM}(J)) \triangleq |\{b \mid j \in J, C_j > 0\}| \tag{2}$$

where $j = (b, F)$ represents a prompt, and C_j is the number of responses that pass the tests for the given prompt. All plausible fixes are tested and manually verified to be the correct fix.

IV. Results and Discussion

A. RQ1: To what extent can LLMs resolve different types of bugs using only immediate bug knowledge?

Results. Table II presents the results of running our entire dataset through the *Bug Knowledge Layer* using GPT-4o-mini and Llama 3.3. When provided only with local context surrounding the buggy function, GPT-4o-mini resolves 197 out of 314 bugs (62%), while Llama slightly outperforms it by fixing 207 bugs (65%). In terms of *Pass@k* performance, both models show consistent capabilities, though Llama is more reliable overall. Llama achieves a *Pass@1* of 47%, compared to 38% for GPT-4o-mini, indicating a 9% higher chance of generating a correct fix on the first attempt. As *k* increases, performance improves for both models: *Pass@3* and *Pass@5* exceed 50%, with Llama reaching 61% at *Pass@5* and maintaining a consistent lead across all values of *k*.

Table III breaks down performance by bug type. Llama performs better on *Network* (e.g., Llama 3.3 resolves 72% bugs compared to 68% by GPT-4o-mini), *Program Anomaly*, *Configuration*, and *Permission/Deprecation* bugs, while GPT-4o-mini shows an advantage for *Performance* bugs. For *GUI*-related bugs, both models fix 16 out of 28 instances (57%), with 13 of those fixes overlapping and 3 unique to each model.

Discussion. Since some of our metrics in this layer are adopted from Parasaram et al. [9], we compare our results to theirs. They achieved a total fix rate of 56% (177/314 bugs), using GPT-3.5 Turbo and Llama3-70B. Our best-performing model in this layer, Llama 3.3, achieves statistically significant improvements over Parasaram et al.'s results, supported by a Chi-square test (*p-value*=0.017) and Z-test (*p-value*=0.014). We exclude two of Parasaram et al.'s metrics, *Buggy Class Declaration* and *Used Method Signatures*, as they introduce information beyond the immediate bug context. The results indicate the strength of our layered approach. Bugs that can be resolved with only bug knowledge are fixed early, without the risk of confusing the model with unnecessary context, while subsequent layers will progressively address bugs that require a deeper understanding of the repository or project (as we will later find out in RQ2 and RQ3). The use of newer LLMs with larger context windows (128k vs. 8k) further supports this approach by enabling the injection of richer knowledge without exceeding model limitations.

Additionally, our bug-type analysis provides actionable guidance for model selection under limited context. For instance, if working with a *Performance* bug and restricted to

TABLE II
PERFORMANCE OF KNOWLEDGE INJECTION LAYERS FOR EACH LLM

Knowledge Injection	LLM	Pass@1	Pass@3	Pass@5	%Fixed Bugs
Bug Knowledge Layer	Llama 3.3	0.47	0.58	0.61	65% (207/314)
	GPT-4o-mini	0.38	0.50	0.56	62% (197/314)
Repository Knowledge Layer	Llama 3.3	0.50	0.63	0.68	74% (235/314)
	GPT-4o-mini	0.40	0.54	0.61	70% (221/314)
Project Knowledge Layer	Llama 3.3	**0.50**	**0.65**	**0.71**	**79% (250/314)**
	GPT-4o-mini	0.40	0.55	0.62	73% (229/314)
All Knowledge Layers at Once	Llama 3.3	0.38	0.53	0.59	65% (207/314)
	GPT-4o-mini	0.27	0.37	0.41	48% (152/314)

bug-level information (e.g., error logs, test failures), GPT-4o-mini may be a better choice. In contrast, for bug types like *Program Anomaly*, Llama shows stronger results. This indicates that the effectiveness of each model can vary depending on the type of bug and the scope of available context. Across all bug types, the Llama model consistently achieves higher *Pass@k* scores, suggesting more reliable performance within fewer attempts compared to GPT-4o-mini. Finally, Llama failed to fix 107 bugs and GPT-4o-mini failed on 117, with 78 overlapping. This intersection suggests that there are particularly challenging cases that LLMs are unable to fix with immediate bug context alone.

B. RQ2: How does injecting repository-level knowledge enhance LLM-based repair of different bug types?

Results. We re-patch the previously unresolved bugs from the *Bug Knowledge Layer* (107 for Llama and 117 for GPT-4o-mini) using updated prompts that inject additional information from the *Repository Knowledge Layer*. The results in Table II show substantial improvements for GPT-4o-mini and a statistically significant improvement for Llama 3.3. GPT-4o-mini successfully resolves 24 additional bugs, increasing its overall fix rate to **70%** (Chi-square *p-value*=0.051 and Z-test *p-value*=0.042), while Llama 3.3 resolves 28 more, reaching **74%** (Chi-square *p-value*=0.018 and Z-test *p-value*=0.014). This represents a significant performance gain of **8–9%** when repository-specific context is layered on top of local bug knowledge. Both models also show consistent improvements in their *Pass@k* scores, with Llama 3.3 continuing to outperform GPT-4o-mini in overall consistency, surpassing the 50% threshold for *Pass@1* in this layer.

As shown in Table III, both models also demonstrate performance improvements across all bug types in the *Repository Knowledge Layer* compared to their respective results in the *Bug Knowledge Layer*. GPT-4o-mini resolves 13 additional *Program Anomaly*, 5 *Network*, 2 *GUI*, 2 *Configuration*, 1 *Performance*, and 1 *Permission/Deprecation* bugs. In comparison, Llama fixes 15 more *Program Anomaly*, 2 *Network*, 4 *GUI*, 3 *Configuration*, and 4 *Performance* bugs, but sees no improvement for *Permission/Deprecation* bugs. It is to be noted that there are only 2 *Permission/Deprecation* bugs in our dataset. This indicates that, regardless of the bug type, adding repository-level context consistently benefits LLM-based repair. While GPT-4o-mini sees greater gains on the number of fixed *Network* bugs, Llama shows stronger improvements overall, outperforming the GPT model across different types of bugs in terms of fix rate and *Pass@k*.

Discussion. These results highlight the role of repository-level knowledge, such as related files, structural dependencies, and recent commit history, in enhancing LLM-based bug repair. When local bug knowledge is insufficient, this additional context allows the model to better understand how the buggy function fits within the broader codebase. The fact that both models improve across bug types suggests that repository knowledge is generically beneficial. Moreover, Llama 3.3 continues to show stronger consistency in this layer, with higher *Pass@k* scores across all bug types. This suggests that it may be better equipped to take advantage of deeper contextual code information when available compared to GPT-4o-mini.

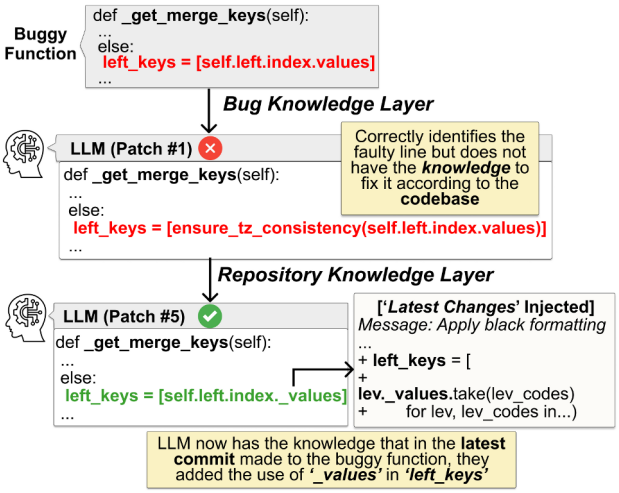

Fig. 3. Comparison of Patch Generation with Repository Knowledge Layer Injected to LLMs. Bug #116 in Project Pandas

To better understand how repository-level information contributes to bug repair, we manually analyzed the additional bugs solved in this layer (28 by Llama and 24 by GPT-4o-mini). For every fixed bug, we examined the structure and logic of the buggy function, the additional context retrieved at the current layer, the generated patch and corresponding test outcomes, and then assessed whether the added context plausibly enabled the fix. Manual analysis was done by the first author, and findings were refined through collaborative thematic analysis with the other authors during group discussions, from which we derive the reported patterns and insights. We found that structural dependencies, related files, and commit history provide crucial missing context that helps LLMs infer the intended behavior of the code and produce accurate fixes. For example (Figure 3), a *Program Anomaly* bug from pandas [49] required changing "*index.values*" to "*index._values*" in one of the lines. In the *Bug Knowledge Layer*, Llama was unaware of "*_values*" as a valid attribute due to its absence in the local context. However, the structural dependencies and the latest commit affecting the function added additional relevant information, and Llama was able to understand the correct usage and generate the right patch.

TABLE III
PERFORMANCE OF KNOWLEDGE INJECTION LAYERS BY BUG TYPE

Bug Type\LLM	GPT-4o-mini									Llama 3.3								
	Bug Knowledge Layer			Repository Knowledge Layer			Project Knowledge Layer			Bug Knowledge Layer			Repository Knowledge Layer			Project Knowledge Layer		
	AVG Pass@1	AVG Pass@3	%Fixed Bugs	AVG Pass@1	AVG Pass@3	%Fixed Bugs	AVG Pass@1	AVG Pass@3	%Fixed Bugs	AVG Pass@1	AVG Pass@3	%Fixed Bugs	AVG Pass@1	AVG Pass@3	%Fixed Bugs	AVG Pass@1	AVG Pass@3	%Fixed Bugs
Program Anomaly	0.34	0.47	60% (113/187)	0.35	0.50	67% (126/187)	0.35	0.50	68% (129/187)	0.41	0.53	62% (117/187)	0.43	0.57	70% (132/187)	**0.43**	**0.59**	**76% (142/187)**
Network	0.43	0.55	68% (32/47)	0.48	0.62	78% (37/47)	0.48	0.63	**80% (38/47)**	0.53	0.65	72% (34/47)	0.57	0.69	76% (36/47)	**0.57**	**0.71**	78% (37/47)
GUI	0.30	0.40	57% (16/28)	0.32	0.46	64% (18/28)	0.37	0.52	75% (21/28)	0.44	0.53	57% (16/28)	0.51	0.64	71% (20/28)	**0.51**	**0.70**	**82% (23/28)**
Configuration	0.48	0.60	70% (24/34)	0.52	0.65	76% (26/34)	0.52	0.65	76% (26/34)	0.68	0.77	85% (29/34)	0.70	0.81	94% (32/34)	**0.70**	**0.82**	**97% (33/34)**
Performance	0.50	0.64	75% (12/16)	0.53	0.70	81% (13/16)	0.53	0.70	81% (13/16)	0.51	0.60	62% (10/16)	**0.58**	**0.74**	**87% (14/16)**	0.58	0.74	87% (14/16)
Permission/ Deprecation	0.0	0.0	0% (0/2)	0.10	0.26	50% (1/2)	0.10	0.26	50% (1/2)	**0.40**	**0.50**	**50% (1/2)**	0.40	0.50	50% (1/2)	0.40	0.50	50% (1/2)
Overall	0.38	0.50	62% (197/314)	0.40	0.54	70% (221/314)	0.40	0.55	73% (229/314)	0.47	0.58	65% (207/314)	0.50	0.63	74% (235/314)	**0.50**	**0.65**	**79% (250/314)**

These patterns extend to other bug types as well. In a *Performance* bug in the luigi project [50], the solution involved setting "*record_task_history=False*" in a call to "*CentralPlannerScheduler*". In the previously generated patches, the LLMs tried to guess what the right call to the class is, even hallucinating non-existent variables such as "*self._config*". Once additional knowledge like the "*CentralPlannerScheduler*" class was injected into the prompt through structural dependency analysis, the LLMs were able to generate correct patches. In a *Network* bug from the fastapi project [51], injecting the definition of the "*HTTPException*" class likely enabled GPT-4o-mini to understand its internal structure and generate correct patches that handle the network error properly. Similarly, in a *GUI* bug from matplotlib [52], the addition of usage examples from co-occurring files along with recent changes to the buggy function provided additional clues about its correct behavior and helped the model identify the underlying issue.

Collectively, these cases show that effective LLM-guided bug repair often needs more than isolated code snippets; it requires situating the buggy function within the larger repository context. Structural and historical insights can provide the missing links that allow LLMs to reason more effectively about how to fix buggy code across different bug types.

C. RQ3: How does injecting project-level knowledge further enhance LLM-based repair of different bug types?

Results. The third row of Table II shows the results after fixing the unresolved bugs after the *Repository Knowledge Layer* using additional project-level knowledge in the *Project Knowledge Layer* (79 unresolved bugs for Llama and 93 for GPT). With this final layer of knowledge injection, Llama 3.3 successfully fixes 15 more bugs, increasing its overall fix rate to **79%**, **5%** more than the previous layer. GPT-4o-mini shows a **3%** improvement, fixing 7 additional bugs and reaching a **73%** fix rate. Both models show marginal gains in *Pass@1*, while improvements are more noticeable in *Pass@3* and *Pass@5*, with 2–3% increases for each. This suggests that, while project-level knowledge may not frequently enable models to solve a bug on the first attempt, it increases the likelihood of success with subsequent attempts. This is especially important for agentic APR workflows where fixes are often generated through iterative attempts [53], [54]. While project context is not a silver bullet, it can boost a model's ability to converge on a correct fix when earlier layers fail.

The impact of project-level knowledge varies across bug types. As shown in Table III, Llama sees improvements in 10 additional *Program Anomaly*, 1 *Network*, 3 *GUI*, and 1 *Configuration* bug. GPT-4o-mini, in comparison, fixes 3 more *Program Anomaly*, 1 *Network*, and 3 *GUI* bugs. Neither model shows any improvement on *Performance* or *Permission/Deprecation* bugs in this layer.

Discussion. Project-related knowledge, such as documentation and past resolved issues, offers nuanced, high-level context that can be valuable for certain types of bugs that require understanding of API behavior, developer intent, or project-specific usage constraints. However, its benefits seem more limited for bugs requiring low- to mid-level understanding of the system, such as performance tuning. Moreover, the effectiveness of this knowledge depends heavily on how well a project maintains its documentation and issue reporting.

To better understand the impact of the *Project Knowledge Layer*, we manually analyzed the newly fixed bugs in this layer, 15 for Llama 3.3 and 7 for GPT-4o-mini, and examined the generated patches. Our analysis followed the same methodology outlined in RQ2, examining how the additional project-level context contributed to successful repair. This was done by the first author, and the findings were refined through collaborative discussions with all authors. For example, in a *Program Anomaly* in the youtube-dl project [55] (Figure 4), the root cause was a regex pattern failing to handle edge cases like integers beginning with zero. Our documentation queries returned a precise constraint: "*The function must recognize and convert integers in both decimal and hexadecimal (prefixed with 0x or 0X) and octal (prefixed with 0)*". Combined with the buggy function and relevant API context, this likely enabled the model to generate correct patches.

We have noticed similar behaviors for other bugs. In a *Network* bug from scrapy [56], the issue involved handling requests to "*robots.txt*" that triggered a "*KeyError*". Past resolved issues were likely useful here. One prior issue described a similar error scenario with the recommendation: "*Implement better error handling to ensure that the function handles cases where netloc might not exist in self._parsers*". With this context, the model correctly applied error handling to resolve the bug. A similar benefit was observed in a *Configuration*

bug [57], where the fix required adjusting how command arguments were preserved. Information from past resolved issue threads emphasizing concerns such as "Correct Output Handling", particularly around capturing error messages, and suggestions like "Preservation of Original Command Arguments" likely guided the model toward a fix.

These examples show that project-level context fills in semantic gaps left by local and repository layers. When well-curated, this layer enables LLMs to reason at a higher level, aligning fixes not just with code structure but with project-specific behaviors and expectations.

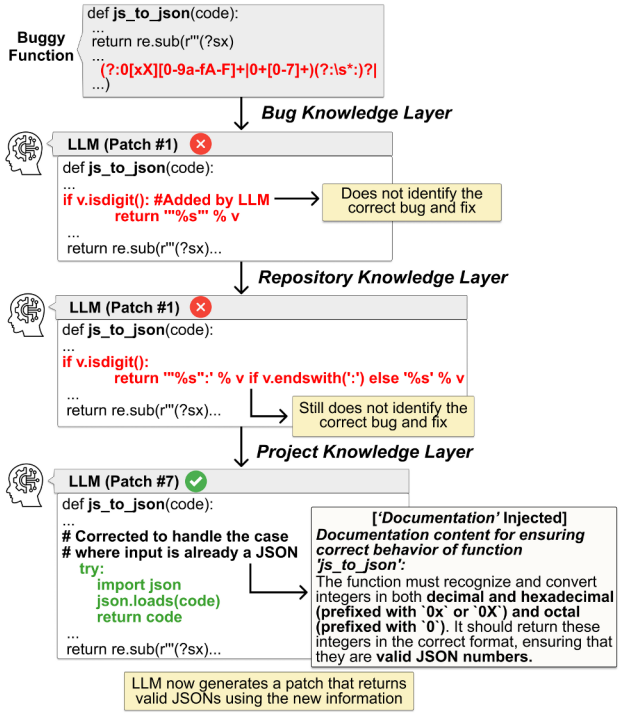

Fig. 4. Comparison of Patch Generation with Different Layers Injected to LLMs for Bug #26 in Project Youtube-dl

In addition to our hierarchical injection strategy, we compare our approach with an additional baseline where contextual knowledge from all three layers is injected simultaneously, i.e., at the same time, for each bug. As shown in the last two rows of Table II), both LLMs perform worse for all *Pass@k* metrics when provided with all the layers' information at once. In terms of the overall percentage of bugs fixed, Llama 3.3 has a fix rate of 65%, identical to its performance using only the first layer, but inferior to the hierarchical configurations that use the repository knowledge and then the project knowledge. GPT-4o-mini performs substantially worse: it fixes 14% fewer bugs compared to its bug-level configuration. These results suggest that providing all information upfront may prevent the model from distinguishing the most relevant context information for a bug, leading to degraded performance. This further highlights the advantage of our hierarchical approach, which allows the model to solve simpler bugs early using minimal context and reserves broader information for more context-hungry bugs that truly require it.

D. RQ4: Which bug types remain challenging for LLMs, even after injecting all layers of contextual knowledge?

Results. After applying all layers of contextual knowledge, GPT-4o-mini still fails to fix 85 bugs, consisting of 58 *Program Anomaly*, 9 *Network*, 7 *GUI*, 8 *Configuration*, 3 *Performance*, and 1 *Permission/Deprecation* bug. Llama leaves 64 bugs unresolved, consisting of 45 *Program Anomaly*, 10 *Network*, 5 *GUI*, 1 *Configuration*, 2 *Performance*, and 1 *Permission/Deprecation* bug. In total, the union of unresolved bugs across both models results in 99 unique cases, with 50 bugs overlapping, indicating a shared set of challenging bugs that neither model could solve. We analyze the union of all unresolved bugs across both models (99 bugs).

First, we report how many of the five additional pieces of context at the repository-level (*Structural Dependencies, Co-occurring Files, Latest Changes*) and project-level (*Documentation, Issue Resolution History*) were available and thus successfully injected for each unresolved bug (Table IV). Of the 99 bugs, 39 received all five pieces of information, 40 were missing one, 18 were missing two, and 2 were missing three. Among the missing elements, *Co-occurring Files* and *Structural Dependencies* were the most frequently absent, suggesting that many of these buggy functions are isolated, either not calling or being called by other parts of the codebase. The same pattern is also observed when looking at the intersection of the unresolved bugs between the two models. Out of the 50 overlapping bugs, only 20 bugs have all the information, 21 were missing one, 8 were missing two, and 1 was missing three, with *Structural Dependencies* and *Co-occurring Files* the most frequently absent information.

TABLE IV
MISSING INFORMATION IN REPOSITORY AND PROJECT LAYERS FOR UNRESOLVED BUGS

Unresolved Bugs In both Models	Have All Information	Missing 1 Information	Missing >1 Information
Union (N=99)	39 (39%)	40 (41%)	20 (20%)
Intersection (N=50)	20 (40%)	21 (42%)	9 (18%)

Table V summarizes the comparison of unresolved bugs and fixed bugs by complexity metrics. Unresolved bugs consistently exhibit higher complexity across all metrics. To quantify the magnitude of difference, we report Cohen's d. For example, *Cyclomatic Complexity* shows a Cohen's d of –0.24, indicating a small to medium effect size with higher complexity in unresolved bugs. Among these, the difference in *Lines of Code* is statistically significant (p-value=0.02), while *Cyclomatic Complexity* and *Halstead Difficulty* are borderline significant with p-values of 0.05 and 0.06, respectively. The *Maintainability Index* is higher for fixed bugs, but the difference is not statistically significant based on the test.

Discussion. Unresolved bugs are not randomly distributed; they cluster around specific bug types and higher complexity profiles. In particular, *Program Anomaly*, *Network*, and *GUI*

bugs remain the most challenging for both models, even after all layered contextual knowledge is injected. These categories often require a deeper understanding of data flow, or even network and user-facing behavior that is not always available. There is a level of interaction and feedback needed, especially in the case of *GUI* and *Network* bugs, to guide the generation of plausible patches with LLMs. In many cases, the buggy function is also too isolated to connect with useful information elsewhere in the codebase. Over 66% of unresolved bugs lacked one or more pieces of injected repository or project-level context, limiting the model's ability to see the full picture. An example of this can be seen in a *GUI* bug from the matplotlib project [58]. The buggy function in this case is 209 lines long, has moderate complexity, and is structurally independent from the rest of the codebase. Fixing a bug at this level likely requires an interactive approach that allows the model to observe the effects of its edits. This showcases the limitations of static prompting and suggests that integrating LLMs into agentic or feedback-driven APR systems, where edits can be tested, observed, and refined, may be essential for handling such bugs.

This observation is further strengthened by our complexity analysis. Unresolved bugs consistently exhibit higher *Cyclomatic Complexity* and *Lines of Code*. This suggests that bugs involving more branching logic, nested control structures, or longer functions may be harder for LLMs to reason about. Higher *Halstead* metrics for unresolved bugs indicate increased cognitive load, reinforcing the idea that LLMs struggle with dense, logic-heavy functions that require precise coordination across multiple code elements. These results highlight a key limitation of current LLMs: while they can pattern-match and generate plausible fixes in simpler scenarios, they fall short when deeper semantic understanding or multi-step reasoning is required. Addressing these limitations will require advances in model reasoning capabilities and how contextual information is retrieved for isolated and specific types of bugs.

TABLE V
COMPLEXITY OF UNRESOLVED BUGS. STATISTICALLY SIGNIFICANT (*)

Metric	Fixed Bugs Mean	Unresolved Bugs Mean	Cohen's d	u-test P-value
Cyclomatic Complexity	9.29	12.14	-0.24 (Unresolved>Fixed)	**0.05**
Lines of Code (LOC)	49.8	65.32	-0.24 (Unresolved>Fixed)	**0.02***
Halstead Volume	136.4	168.34	-0.11 (Unresolved>Fixed)	0.10
Halstead Difficulty	2.63	3.08	-0.15 (Unresolved>Fixed)	**0.06**
Halstead Effort	954.0	1323.63	-0.11 (Unresolved>Fixed)	0.08
Maintainability Index	73.4	72.57	0.05 (Unresolved<Fixed)	0.73

V. THREATS TO VALIDITY

Construct Validity. A threat to construct validity might arise from the manual bug type annotations. To mitigate this threat, we used an iterative coding process including multiple rounds of discussion and conflict resolution. Each annotator has 3+ years of experience in programming and qualitative research. We also observed a high Cohen's Kappa inter-rater agreement of 0.8, indicating strong agreement [19].

Internal Validity. The training data of proprietary LLMs is not fully disclosed. This limits our ability to verify whether specific bugs or repository content existed in the models' pre-training data. However, our study focuses on the relative effectiveness of different knowledge injection layers, not on absolute performance. For example, for a given bug, if the repository knowledge layer leads to a successful fix where the bug knowledge layer does not, the observed improvement is attributed to the additional injected context, regardless of whether the model has seen the bug before. This design isolates the impact of contextual knowledge and mitigates the influence of potential leakage. To further strengthen our analysis, we evaluate two architecturally distinct models (Llama 3.3 and GPT-4o-mini), reducing the likelihood that any results hinge on the behavior of a single LLM. Moreover, a recent investigation into benchmark leakage [59] using the LLM *StarCoder* found only 11% leakage rate in the BugsInPy dataset. While our study does not use *StarCoder*, it is likely that if leakage exists, it is limited and unlikely to dominate outcomes. Lastly, while we curated our layered knowledge based on insights from the APR literature, our layers do not represent an exhaustive set of all potentially useful information. Future work may explore alternative or additional forms of contextual input to further refine LLM-based APR.

External Validity. Our findings may not generalize to all LLMs. However, to improve generalizability, we evaluated two distinct models: Llama 3.3 (70B parameters) and GPT-4o-mini ($\sim$8B parameters). This allows us to test our framework across models with different scales and capabilities [60], and observe consistent trends in performance improvements across knowledge layers. We also acknowledge that the distribution of bug types in our dataset is imbalanced. Underrepresented categories (e.g., *Permission/Deprecation*) may not yield insights that are broadly applicable to those bug types.

VI. RELATED WORK

Prompt Engineering and In-context Learning for APR. Recent research has explored a wide range of techniques to enhance the effectiveness of LLMs in fixing software bugs, including prompt engineering, fine-tuning, and context injection. Xia et al. [61] first showed that prompting LLMs with buggy functions, without any additional context or customization, outperforms traditional APR tools. Building on this, MMAPR [62] and RING [63] applied prompt learning for syntactic and multi-language repair, while InferFix [64] used few-shot prompting to fix static analysis issues. These early systems focused on local bug context and overlooked broader project-level information. The limitations of LLMs became more apparent with the release of the SWE-Bench dataset [65], which includes real-world bugs from large projects. Initial attempts showed low success rates (around 2%), even when LLMs were provided with the localized buggy functions and asked to generate repairs [65]. Ehsani et al. [27], [66] found

that different prompting strategies (e.g., chain-of-thought, tree-of-thought) do not yield substantial improvements in bug resolution tasks, suggesting that merely tweaking prompt structure is not enough. To address this, researchers explored integrating richer context. Fan et al. [67] showed that fault localization signals improve fix accuracy. ChatRepair [23] introduced interactive prompting using failing test names and assertions. Other studies found that including local bug context [1], relevant identifiers [2], [68], and structured cues from stack traces [24], [69] improves repair quality. Contextual cues from bug reports and issue descriptions have also been shown to enhance performance [26], as has incorporating historical signals such as previous blame commits [10]. Building on these ideas, Parasaram et al. [9] proposed MANIPLE, a tool that selects and adds bug-related facts into LLM prompts. Their results showed significant gains over naive prompting, especially when space in the context window was limited.

Agentic and Iterative Workflows for APR. Recent work has moved beyond static prompting toward agent-based [53] and iterative workflows to enhance LLM-based APR. These systems perform multi-round querying, reasoning, and context retrieval to explore large codebases and refine patches [3]. OpenHands [70] combines multi-query prompting with agents in sandbox environments to test and validate fixes. AutoCodeRover [54] uses hierarchical code search to support bug localization and patch generation. Both tools show performance gains on SWE-Bench, highlighting the value of interactive, repository-aware workflows. Iterative approaches have also proven promising. Ruiz et al. [71] used self-iterative prompting, allowing LLMs to refine patches across up to 10 rounds, yielding up to 78% improvement in generating plausible patches. Similarly, Yang et al. enhanced prompt quality and precision by integrating repository-level knowledge graphs [72]. Other works have explored integrating multiple LLMs and tool agents into a unified APR pipeline [73].

Overall, while prior work advanced LLM-based APR, it often (1) limits context to the immediate bug, (2) treats all bugs the same, and (3) relies on black-box agentic methods that lack interpretability. We address these gaps with a transparent, deterministic, layered knowledge injection framework that incrementally adds bug, repository, and project-level context, enabling fine-grained analysis across bug types.

VII. Conclusion and Future Work

We introduced a layered knowledge injection framework for LLM-based APR, addressing key limitations in previous work that focused narrowly on isolated local bug context while ignoring the broader structure of real-world software systems. Our approach iteratively injects *bug*, *repository*, and *project*-level contextual information into prompts, enabling more effective and context-aware repair across a diverse set of bug types. Our results confirm the hypothesis posed in our motivating example: one of the missing pieces in LLM-based APR is a deeper understanding of repository and project context. We observed consistent gains in both *#fixed* and *Pass@k* scores across layers for both Llama 3.3 and GPT-4o-mini, with Llama achieving a 79% fix rate, a significant improvement of 23% over prior work [9]. All bug types showed improvement from *Bug Knowledge Layer* to *Repository Knowledge Layer*, while only a subset (*Program Anomaly*, *Network*, and *GUI*) further benefited from *Project Knowledge Layer*. Injecting all contextual information at once underperformed our layered strategy for all bug types, reinforcing the importance of incremental, context-aware knowledge injection for maximizing repair performance.

Error analysis shows that the remaining unresolved bugs are more complex, structurally isolated, or dependent on particular behavior like user interactions and edge-case reasoning, especially for *Program Anomaly*, *Network*, and *GUI* bugs. Future work will focus on integrating our approach with agentic workflows, such as OpenHands [70], which can actively navigate codebases and incorporate feedback during repair. Combining structured context injection with dynamic, feedback-driven reasoning could lead to more adaptive and effective repair systems. Additionally, project-specific fine-tuning on relevant repositories can further improve model alignment with code conventions and domain-specific behavior, leading to more accurate and reliable repairs in complex software systems.

References

[1] J. A. Prenner and R. Robbes, "Out of context: How important is local context in neural program repair?" in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639086

[2] C. S. Xia, Y. Ding, and L. Zhang, "The plastic surgery hypothesis in the era of large language models," in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '23. IEEE Press, 2024, p. 522–534. [Online]. Available: https://doi.org/10.1109/ASE56229.2023.00047

[3] F. Li, J. Jiang, J. Sun, and H. Zhang, "Hybrid automated program repair by combining large language models and program analysis," *ACM Trans. Softw. Eng. Methodol.*, Jan. 2025, just Accepted. [Online]. Available: https://doi.org/10.1145/3715004

[4] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: combining context-aware neural translation models using ensemble for program repair," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 101–114. [Online]. Available: https://doi.org/10.1145/3395363.3397369

[5] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1–11. [Online]. Available: https://doi.org/10.1145/3180155.3180233

[6] A. Chen, H. Wu, Q. Xin, S. P. Reiss, and J. Xuan, "Studying and understanding the effectiveness and failures of conversational llm-based repair," 2025. [Online]. Available: https://arxiv.org/abs/2503.15050

[7] J. Zhao, D. Yang, L. Zhang, X. Lian, Z. Yang, and F. Liu, "Enhancing automated program repair with solution design," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1706–1718. [Online]. Available: https://doi.org/10.1145/3691620.3695537

[8] J. Zhao, Z. Yang, L. Zhang, X. Lian, D. Yang, and X. Tan, "Drminer: Extracting latent design rationale from jira issue logs," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 468–480. [Online]. Available: https://doi.org/10.1145/3691620.3695019

[9] N. Parasaram, H. Yan, B. Yang, Z. Flahy, A. Qudsi, D. Ziaber, E. T. Barr, and S. Mechtaev, "The fact selection problem in llm-based program repair," in *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, ser. ICSE '25. IEEE Press, 2025, p. 2574–2586.

[10] Y. Shi, A. A. Bangash, E. Fallahzadeh, B. Adams, and A. E. Hassan, "Hafix: History-augmented large language models for bug fixing," 2025. [Online]. Available: https://arxiv.org/abs/2501.09135

[11] D. Hao, L. Zhang, H. Zhong, H. Mei, and J. Sun, "Eliminating harmful redundancy for testing-based fault localization using test suite reduction: an experimental study," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*, 2005, pp. 683–686.

[12] G. Catolino, F. Palomba, A. Zaidman, and F. Ferrucci, "Not all bugs are the same: Understanding, characterizing, and classifying bug types," *Journal of Systems and Software*, vol. 152, pp. 165–181, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121219300536

[13] R. Widyasari, S. Q. Sim, C. Lok, H. Qi, J. Phan, Q. Tay, C. Tan, F. Wee, J. E. Tan, Y. Yieh, B. Goh, F. Thung, H. J. Kang, T. Hoang, D. Lo, and E. L. Ouh, "Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 1556–1560. [Online]. Available: https://doi.org/10.1145/3368089.3417943

[14] J. Aranda and G. Venolia, "The secret life of bugs: Going past the errors and omissions in software repositories," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. USA: IEEE Computer Society, 2009, p. 298–308. [Online]. Available: https://doi.org/10.1109/ICSE.2009.5070530

[15] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, "How do professional developers comprehend software?" in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 255–265.

[16] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Transactions on Software Engineering*, vol. 32, no. 12, pp. 971–987, 2006.

[17] O. Ovadia, M. Brief, M. Mishaeli, and O. Elisha, "Fine-tuning or retrieval? comparing knowledge injection in LLMs," in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, Y. Al-Onaizan, M. Bansal, and Y.-N. Chen, Eds. Miami, Florida, USA: Association for Computational Linguistics, Nov. 2024, pp. 237–250.

[18] K. Liu, A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, and Y. Le Traon, "You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, 2019, pp. 102–113.

[19] M. McHugh, "Interrater reliability: The kappa statistic," *Biochemia medica : časopis Hrvatskoga društva medicinskih biokemičara / HDMB*, vol. 22, pp. 276–82, 10 2012.

[20] T. Ahmed and P. Devanbu, "Better patching using llm prompting, via self-consistency," in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '23. IEEE Press, 2024, p. 1742–1746.

[21] U. Kulsum, H. Zhu, B. Xu, and M. d'Amorim, "A case study of llm for automated vulnerability repair: Assessing impact of reasoning and patch validation feedback," ser. AIware 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 103–111. [Online]. Available: https://doi.org/10.1145/3664646.3664770

[22] J. Li, G. Li, Y. Li, and Z. Jin, "Structured chain-of-thought prompting for code generation," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 2, Jan. 2025. [Online]. Available: https://doi.org/10.1145/3690635

[23] C. S. Xia and L. Zhang, "Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using chatgpt," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 819–831. [Online]. Available: https://doi.org/10.1145/3650212.3680323

[24] J. Keller and J. Nowakowski, "Ai-powered patching: the future of automated vulnerability fixes," Tech. Rep., 2024.

[25] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 691–701.

[26] S. Fakhoury, S. Chakraborty, M. Musuvathi, and S. K. Lahiri, "Nl2fix: Generating functionally correct code edits from bug descriptions," in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, ser. ICSE-Companion '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 410–411. [Online]. Available: https://doi.org/10.1145/3639478.3643526

[27] R. Ehsani, S. Pathak, and P. Chatterjee, "Towards detecting prompt knowledge gaps for improved llm-guided issue resolution," in *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*, 2025, pp. 699–711. [Online]. Available: https://doi.org/10.1109/MSR66628.2025.00107

[28] X. B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, 2016, pp. 213–224.

[29] N. Ajienka, A. Capiluppi, and S. Counsell, "An empirical study on the interplay between semantic coupling and co-change of software classes," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1791–1825, Jun. 2018. [Online]. Available: https://doi.org/10.1007/s10664-017-9569-2

[30] E. Hrishikesh, A. Kumar, M. Bhardwaj, and S. Agarwal, "Co-change graph entropy: A new process metric for defect prediction," 2025. [Online]. Available: https://arxiv.org/abs/2504.18511

[31] Y. Zhang, G. Li, Z. Jin, and Y. Xing, "Neural program repair with program dependence analysis and effective filter mechanism," 2023.

[32] S. Dikici and T. T. Bilgin, "Advancements in automated program repair: a comprehensive review," *Knowledge and Information Systems*, Mar. 2025. [Online]. Available: https://doi.org/10.1007/s10115-025-02383-9

[33] David.Halter, "jedi: An autocompletion tool for Python that can be used for text editors." 11 2024. [Online]. Available: https://github.com/davidhalter/jedi

[34] Y. Shi, A. A. Bangash, E. Fallahzadeh, B. Adams, and A. E. Hassan, "Hafix: History-augmented large language models for bug fixing," 2025.

[35] S. Wijaya, J. Bolano, A. G. Soteres, S. Kode, Y. Huang, and A. Sahai, "Readme.llm: A framework to help llms understand your library," 2025. [Online]. Available: https://arxiv.org/abs/2504.09798

[36] X. Ren, J. Sun, Z. Xing, X. Xia, and J. Sun, "Demystify official api usage directives with crowdsourced api misuse scenarios, erroneous code examples and patches," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 925–936. [Online]. Available: https://doi.org/10.1145/3377811.3380430

[37] W. Wang, Y. Wang, S. Joty, and S. C. Hoi, "Rap-gen: Retrieval-augmented patch generation with codet5 for automatic program repair," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 146–158. [Online]. Available: https://doi.org/10.1145/3611643.3616256

[38] A. Patil, "Gitbugs: Bug reports for duplicate detection, retrieval augmented generation, triage, and more," 2025. [Online]. Available: https://arxiv.org/abs/2504.09651

[39] M. Douze, A. Guzhva, C. Deng, J. Johnson, G. Szilvasy, P.-E. Mazaré, M. Lomeli, L. Hosseini, and H. Jégou, "The faiss library," 2024.

[40] LangChain, "LangChain," 2025. [Online]. Available: https://www.langchain.com/

[41] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, M. Monperrus, J. Klein, and Y. Le Traon, "ifixr: bug report driven program repair," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 314–325.

[42] M. Motwani and Y. Brun, "Better automatic program repair by using bug reports and tests together," 2023. [Online]. Available: https://arxiv.org/abs/2011.08340

[43] radon, "radon: Code Metrics in Python." [Online]. Available: https://radon.readthedocs.org/

[44] F. F. Xu, B. Vasilescu, and G. Neubig, "In-ide code generation from natural language: Promise and challenges," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 2, Mar. 2022.

[45] R. Sandouka and H. Aljamaan, "Python code smells detection using conventional machine learning models," *PeerJ Computer*

Science, vol. 9, p. e1370, May 2023. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC10280480/

[46] P. Ranganathan, "An Introduction to Statistics: Choosing the Correct Statistical Test," *Indian Journal of Critical Care Medicine : Peer-reviewed, Official Publication of Indian Society of Critical Care Medicine*, vol. 25, no. Suppl 2, pp. S184–S186, May 2021. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC8327789/

[47] G. M. Sullivan and R. Feinn, "Using Effect Size—or Why the P Value Is Not Enough," *Journal of Graduate Medical Education*, vol. 4, no. 3, pp. 279–282, Sep. 2012. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3444174/

[48] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 24–36. [Online]. Available: https://doi.org/10.1145/2771783.2771791

[49] GitHub, "Pandas bug 116," 2019. [Online]. Available: https://github.com/pandas-dev/pandas/commit/c4fa6a52f7737aecda08f6b0f2d6c27476298ae1

[50] ——, "Luigi bug 17," 2016. [Online]. Available: https://github.com/spotify/luigi/commit/e38392a1381dd8daee0f180f0ac7f651edb88e0c

[51] ——, "Fastapi bug 12," 2019. [Online]. Available: https://github.com/fastapi/fastapi/commit/d262f6e9296993e528e2327f0a73f7bf5514e7c6

[52] ——, "Matplotlib bug 30," 2019. [Online]. Available: https://github.com/matplotlib/matplotlib/commit/d4de838fe7b38abb02f061540fd93962cc063fc4

[53] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "Swe-agent: Agent-computer interfaces enable automated software engineering," 2024.

[54] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "Autocoderover: Autonomous program improvement," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 1592–1604.

[55] GitHub, "Youtubedl bug 26," 2016. [Online]. Available: https://github.com/ytdl-org/youtube-dl/commit/47212f7bcbd59af40f91796562a6b72ba0439ac4

[56] ——, "Scrapy bug 21," 2016. [Online]. Available: https://github.com/scrapy/scrapy/commit/a8a6f050e71fbb7881076a8d6e2867e868d26016

[57] ——, "Thefuck bug 11," 2016. [Online]. Available: https://github.com/nvbn/thefuck/commit/db7dffdb44ae5c7be8de088765463fbda96197d1

[58] ——, "Matplotlib bug 2," 2020. [Online]. Available: https://github.com/matplotlib/matplotlib/commit/d86cc2bab8183fd3288ed474e4dfd33e0f018908

[59] X. Zhou, M. Weyssow, R. Widyasari, T. Zhang, J. He, Y. Lyu, J. Chang, B. Zhang, D. Huang, and D. Lo, "Lessleak-bench: A first investigation of data leakage in llms across 83 software engineering benchmarks," 2025. [Online]. Available: https://arxiv.org/abs/2502.06215

[60] S. Banerjee, M. Sussman, and Y. Lian, "Dimensional Analysis in Error Reduction for Prediction of Nucleate Boiling Heat Flux by Artificial Neural Networks for Limited Dataset," *ASME Journal of Heat and Mass Transfer*, vol. 145, no. 6, p. 061602, 02 2023. [Online]. Available: https://doi.org/10.1115/1.4056539

[61] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE '23. IEEE Press, 2023, p. 1482–1494.

[62] J. Zhang, J. P. Cambronero, S. Gulwani, V. Le, R. Piskac, G. Soares, and G. Verbruggen, "Pydex: Repairing bugs in introductory python assignments using llms," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA1, Apr. 2024.

[63] H. Joshi, J. C. Sanchez, S. Gulwani, V. Le, I. Radiček, and G. Verbruggen, "Repair is nearly generation: multilingual program repair with llms," in *Proceedings of the Thirty-Seventh AAAI Conference on Artificial Intelligence and Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence and Thirteenth Symposium on Educational Advances in Artificial Intelligence*, ser. AAAI'23/IAAI'23/EAAI'23. AAAI Press, 2023.

[64] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy, "Inferfix: End-to-end program repair with llms," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 1646–1656.

[65] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. R. Narasimhan, "Swe-bench: Can language models resolve real-world github issues?" in *ICLR*, 2024.

[66] R. Ehsani, S. Pathak, E. Parra, S. Haiduc, and P. Chatterjee, "What characteristics make chatgpt effective for software issue resolution? an empirical study of task, project, and conversational signals in github issues," *Empirical Software Engineering*, 2025, in press. [Online]. Available: https://arxiv.org/abs/2506.22390

[67] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. H. Tan, "Automated repair of programs from large language models," in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE '23. IEEE Press, 2023, p. 1469–1481. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00128

[68] N. Jiang, K. Liu, T. Lutellier, and L. Tan, "Impact of code language models on automated program repair," in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE '23. IEEE Press, 2023, p. 1430–1442.

[69] M. Haque, P. Babkin, F. Farmahinifarahani, and M. Veloso, "Towards effectively leveraging execution traces for program repair with code LLMs," in *Proceedings of the 4th International Workshop on Knowledge-Augmented Methods for Natural Language Processing*, W. Shi, W. Yu, A. Asai, M. Jiang, G. Durrett, H. Hajishirzi, and L. Zettlemoyer, Eds. Albuquerque, New Mexico, USA: Association for Computational Linguistics, May 2025, pp. 160–179.

[70] X. Wang, B. Li, Y. Song, F. F. Xu, X. Tang, M. Zhuge, J. Pan, Y. Song, B. Li, J. Singh, H. H. Tran, F. Li, R. Ma, M. Zheng, B. Qian, Y. Shao, N. Muennighoff, Y. Zhang, B. Hui, J. Lin, R. Brennan, H. Peng, H. Ji, and G. Neubig, "Openhands: An open platform for ai software developers as generalist agents," 2025. [Online]. Available: https://arxiv.org/abs/2407.16741

[71] F. V. Ruiz, M. Hort, and L. Moonen, "The art of repair: Optimizing iterative program repair with instruction-tuned models," 2025. [Online]. Available: https://arxiv.org/abs/2505.02931

[72] B. Yang, H. Tian, J. Ren, S. Jin, Y. Liu, F. Liu, and B. Le, "Enhancing repository-level software repair via repository-aware knowledge graphs," 2025. [Online]. Available: https://arxiv.org/abs/2503.21710

[73] J. Zhao, D. Yang, L. Zhang, X. Lian, Z. Yang, and F. Liu, "Enhancing automated program repair with solution design," 2024. [Online]. Available: https://arxiv.org/abs/2408.12056