How Do LLMs Read Bug Reports? An Empirical Study of Attention in LLMs for Automated Program Repair

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

Irene Manotas
irene.manotas@ibm.com
IBM Research
Yorktown, NY, USA

Saurabh Pujar
saurabh.pujar@ibm.com
IBM Research
Yorktown, NY, USA

Luca Buratti
luca.buratti1@ibm.com
IBM Research
Yorktown, NY, USA

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

arXiv:2607.25873v1 [cs.SE] 28 Jul 2026

Abstract

Large Language Model (LLM)-based Automated Program Repair systems are advancing rapidly, yet their performance remains inconsistent. Even when provided with the same contextual information, an LLM may generate a correct patch for one bug but fail on another closely related bug. Why this happens remains poorly understood, and it is unclear how LLMs prioritize the diverse information in bug reports and whether model attention affects repair success.

In this paper, we present the first empirical study of attention patterns in LLM-based program repair, providing interpretable insights into how models process bug reports and where their attention is concentrated during repair. We analyze 319 real-world Python and Java bugs from *SWE-bench Verified* and *Multi-SWE-bench* to study **(RQ1)** how model attention is distributed across bug report sections, **(RQ2)** how attention patterns within each section differ between successful and unsuccessful repairs, and **(RQ3)** how these patterns compare to information developers consider important for bug fixing. We find that successful repairs are characterized by diffused attention across multiple diagnostic components such as bug descriptions, stacktraces, and test cases, while failures often exhibit over-localized attention toward metadata such as version information. We further observe that stronger alignment between model attention and developer-identified key sections and phrases is associated with higher repair success. Our results provide the first empirical evidence that attention misallocation is a key factor in LLM-based APR failures, and offer actionable insights for designing more interpretable and reliable future APR systems.

CCS Concepts

• **Software and its engineering → Software maintenance tools**; **Automatic programming**; • **Computing methodologies → Artificial intelligence**.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference'17, Washington, DC, USA
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Keywords

large language models, attention analysis, program repair

ACM Reference Format:
Ramtin Ehsani, Irene Manotas, Saurabh Pujar, Luca Buratti, and Preetha Chatterjee. 2026. How Do LLMs Read Bug Reports? An Empirical Study of Attention in LLMs for Automated Program Repair. In . ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

LLMs are now deeply embedded in software development tasks such as Automated Program Repair (APR), which involves automatically generating code patches to fix software bugs [82]. Given a bug report and the corresponding buggy code, LLMs attempt to produce a patch that resolves the bug while preserving the intended program behavior. Although LLM-based program repair has shown significant improvements over traditional techniques [12, 77, 78], failure cases remain frequent and unpredictable [49]. A model may successfully fix one bug while failing on a nearly identical one, providing no insight into its decision-making process [12, 43]. This lack of interpretability limits our ability to understand and integrate LLM-based repair systems into developer workflows.

Bug reports contain multiple sections, such as a bug description, steps to reproduce the failure, expected behavior, and other information. Repairing real bugs is a complex task, particularly for LLMs, which must attend to the right type and amount of information to succeed [12]. To generate a correct patch, models must infer the underlying cause of failure, reason about the intended behavior, and integrate signals distributed across multiple sections of the bug report. Recent studies suggest that errors in code generation tasks often stem from how LLMs allocate attention to different types of input information [11, 39, 84, 85]. This raises the question of whether a similar phenomenon also explains failures in LLM-based program repair. While prior studies have analyzed model attention for code generation from short task descriptions [39, 46, 59, 62], extending such analysis to real-world bug reports is substantially more challenging because bug reports are often long, heterogeneous, and contain both code and natural-language information. Understanding where models focus their attention can reveal which parts of a bug report most influence repair decisions.

In this paper, we present the first empirical study of how LLMs allocate attention to bug reports during automated program repair. We analyze 319 Python and Java bugs with different levels

of difficulty (*Easy*, *Medium*, and *Hard*) from *SWE-bench Verified* and *Multi-SWE-bench* using both proprietary (claude-4-sonnet [2]) and open-source (gpt-oss-20b [18], qwen-3-32b [15]) LLMs. Using perturbation-based analysis, we measure how LLMs' attention is distributed across bug report sections and investigate how these patterns differ between successful and unsuccessful repairs. Additionally, we examine whether LLMs attend to the same sections and fine-grained phrases that developers consider most important for repair. Specifically, we investigate the following questions:

- **RQ1:** *How is model attention allocated across bug report sections in successful and unsuccessful repairs?* We examine section-level attention using perturbation-based analysis and find that successful repairs consistently prioritize *Bug description* while unsuccessful repairs over-attend to *Version information*.
- **RQ2:** *How is model attention distributed across specific code and natural-language components within bug report sections?* We identify attention patterns that differentiate repair outcomes using perturbations on natural language and code components within reports, showing that successful repairs rely on *diffused attention* across important information such as stacktrace and test data, while failures often arise from overly *localized attention* on contextual metadata such as library versions.
- **RQ3:** *How well do model attention patterns align with what developers consider important for bug-fixing?* Human developers rely on experience-driven intuition to identify the most relevant parts of a bug report during debugging [6, 7, 38, 90], but existing APR benchmarks do not capture this information. To fill this gap, we create the first developer attention dataset on a subset of 100 bug reports. Comparing developer annotations with model attention, we show that stronger developer-model attention alignment is significantly associated with successful repairs.

Our findings provide new insights into LLM-based APR systems by revealing where models focus when processing bug reports and how specific attention patterns relate to repair success and failure. These insights can help practitioners design more effective LLM-APR workflows and guide researchers in developing systems that better prioritize diagnostically important information.

The main contributions of this paper are as follows: (1) We present the first empirical study of attention patterns in LLM-based APR, analyzing how models attend to bug reports during repair. (2) We characterize both section-level and fine-grained attention patterns across 319 Python and Java bugs, showing how attention allocation differs between successful and unsuccessful repairs. (3) We evaluate human-model attention alignment by comparing model attention with the information developers consider most important for bug repair. (4) We present the first annotated developer attention dataset of 100 bug reports for the task of APR.

2 Background and Motivation

Consider the example in Figure 1, which shows two closely related but distinct GUI bugs #16344 and #16420 from the *matplotlib* project on GitHub. These two bugs occur within a few months of each other, affect the same function (`'nonsingular'`) in the same file related to the `'colorbar'` component, and require the same fix (link1 and link2). However, their bug reports point to the same underlying issue in slightly different ways. Following prior work [12], we prompt GPT-4o-mini to repair each bug using the same prompt template and model settings, with their corresponding bug reports. The model successfully generates a correct patch for bug #16420 but fails to repair bug #16344. ***What drives divergent repair outcomes when the same model is given comparable information for two closely related bugs?*** We cannot answer because we do not know which parts of the bug reports the model attends to and what information it prioritizes during repair, or how this differs between the case where it succeeds and where it fails.

A closer look at the bug reports suggests a potential explanation. Bug #16420 provides a clear description of the failure (e.g., `'TypeError'`) along with a reproducible example and an explicit workaround. Bug #16344 describes the issue in a more indirect way, explaining value ranges without clearly isolating the failure. While both reports contain the necessary information for developers to solve the bugs, they differ in how that information is presented and, therefore, in which parts the model may focus on during repair.

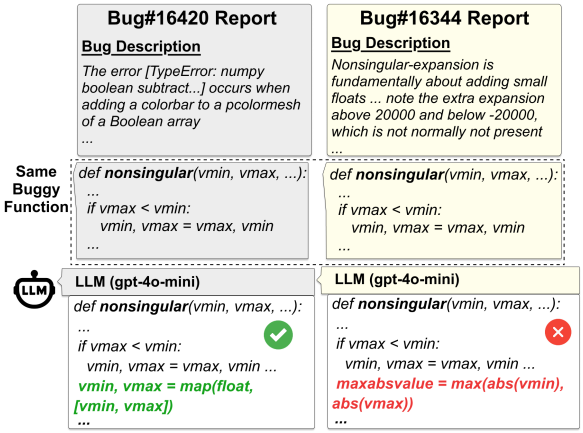

Figure 1: Generated patches for bug instances *matplotlib#16344* and *#16420*.

This behavior is not unique to this example. Such inconsistencies are frequently observed in LLM-based program repair. Even when models are provided with similar information, their repair performance can vary substantially across seemingly similar types of bugs [12–14, 63]. This unpredictability is widely recognized as one of the central challenges when applying LLMs in practice [17, 19, 25, 74].

One way to address this challenge is to understand how models attend to different information in the input through **attention analysis** [75]. Prior work has examined internal model behavior, such as attention heads across layers [9, 28, 57, 58, 65, 75], as well as gradient-based signals that estimate which input components most influence predictions [3, 28]. However, directly studying attention within LLM architectures is often infeasible, as many proprietary models do not expose their internal mechanisms. Even with open-source access, analyzing attention in large models can still be computationally expensive.

Perturbation-based attention analysis offers a practical alternative. Rather than inspecting internal model states or transformer self-attention directly, these methods modify parts of the input and measure how the output changes under deterministic decoding.

If removing a portion of the input significantly alters the generated output, that component likely played an important role in the model's decision [36, 39, 53, 72]. This motivates our study design: we systematically remove information from bug reports and observe the resulting changes in generated repairs to identify which elements LLMs prioritize during program repair. Perturbation-based approaches are model-agnostic and can therefore be applied to both proprietary and open-source LLMs. Prior work in software engineering has also shown that perturbation-based methods align well with human reasoning when compared to other methods such as self-attention or gradient-based analysis [39, 53].

3 Methodology

Dataset. We analyze a total of 319 bugs from two widely used APR benchmarks created from real-world GitHub projects: *SWE-bench Verified* [31] for Python, and *Multi-SWE-bench* [16] for Java. We focus on Python and Java because they are among the most commonly used languages in software projects [70]. Including both allows us to evaluate whether attention patterns are consistent across different languages. Because our goal is to analyze attention during patch generation, we adopt *function-level perfect fault localization* [12, 52, 63]. Under this setting, the buggy function is given to the model, but it must still identify and repair the faulty lines within it. We manually inspect both benchmarks and retain only single-function bugs, resulting in 248 Python and 71 Java bugs.

Both benchmarks are manually curated by their original authors to include only the bugs that are sufficiently described and solvable by LLMs [16, 31]. This is essential for our study because we aim to understand how attention behaves when relevant information is present, and why some repairs succeed while others fail. In addition, both datasets provide a manually annotated difficulty label for each bug (*Easy*, *Medium*, or *Hard*). Across our dataset, 144 bugs are labeled *Easy*, 164 *Medium*, and 11 *Hard*. The distribution is therefore dominated by easy and medium bugs, with relatively few hard instances. We later use these difficulty annotations to examine whether the observed attention patterns remain consistent after accounting for bug difficulty.

After manually examining all 319 bug reports, we observed that they consistently contain several common structured sections. We group these sections into the following categories [44, 68]: 1) **Bug description**, a description of the symptoms of the bug; 2) **Reproduction**, steps or instructions to trigger the failure; 3) **Expected behavior**, a description of the intended or correct system behavior; 4) **Actual behavior**, a description of the current system behavior; 5) **Version information**, has information on the specific version of system/tools used when the bug occurs; and 6) **Additional information**, any supporting details not captured that are relevant to the bug. Not all bug reports contain every section. However, when present, they consistently fall into these categories. Across our dataset, *Bug description* appears in all bug reports, *Reproduction* in 133, *Version information* in 98, *Expected behavior* in 97, *Additional information* in 62, and *Actual behavior* in 41. These sections form the basis for our section-level and phrase-level attention analyses.

Models. We study a combination of proprietary and open-source LLMs. For proprietary models, we analyze *claude-4-sonnet* because of its state-of-the-art performance on coding benchmarks [2]. For open-source models, we use *gpt-oss-20b* and *qwen3-32b*, as they are repeatedly cited among the most capable open-source models for coding [15, 18]. Studying both sets of models allows us to analyze attention patterns in both commercial and open-source models. We also include both small and large models to see whether attention patterns are consistent across scales.

In *RQ1*, we perform section-level attention analysis across all models to obtain a comparative view of how different LLMs distribute attention over bug report sections. In contrast, the perturbation analyses for *RQ2* and *RQ3* are substantially more expensive, as they require masking each component in a bug report and regenerating a patch for every perturbation. Across 319 bugs, this would result in 2,873 perturbations and a large number of model executions. We therefore focus our *RQ2* and *RQ3* analysis for *qwen3-32b* only. We select *qwen3-32b* because it is a strong open-source LLM suitable for multi-lingual coding and is often used as the backbone of several APR systems [1, 15, 26, 42, 80]. To assess the generality of our findings, we additionally repeat the RQ2 analysis on the Java subset (71 bugs) of our dataset using *gpt-oss-20b* to see if it exhibits the same overall attention trends observed with *qwen3-32b*.

For all models, patches are generated using deterministic decoding. We set the temperature to zero and disable reasoning modes when available. This reduces sampling variability and ensures that differences in outputs are primarily due to prompt perturbations.

Prompts. All prompts use a standardized template designed for program repair [12, 63]. Each prompt begins with high-level instructions describing the repair task, followed by the buggy function to be fixed, and finally the full bug report. We use this standardized prompt template across all models to reduce confounding factors introduced by prompt structure variations. The detailed prompt template is provided in our replication package [61].

3.1 Research Questions

3.1.1 RQ1: How is model attention allocated across bug report sections in successful and unsuccessful repairs? We first compare repair performance across models and bug difficulty levels. To better understand successful and unsuccessful repairs, we compare all patches generated by each model against the corresponding ground-truth developer patches using the CodeBLEU similarity metric [64].

Then, to examine whether repair success depends on which sections the model prioritizes when generating a patch, we use perturbation-based attention analysis. Specifically, we consider each bug report as a set of six distinct sections: *Bug description*, *Reproduction*, *Expected behavior*, *Actual behavior*, *Version Information*, and *Additional information*. Each section is treated as a feature for perturbation that can be either *kept* or *masked*. We identify these sections using the headers present in the markdown of each report.

Following prior work [39], we define a binary mask $\mathbf{z} \in \{0,1\}^m$ over the m sections of a bug report, where z_i=1 indicates that section i is removed. For each mask $\mathbf{z}$, we regenerate a patch and measure how much it differs from the patch generated using the full report, i.e., $\mathbf{z} = \mathbf{0}$. To estimate the contribution of each section, we apply Kernel SHAP over coalitions of masked sections [5, 54], using CodeBLEU similarity [64] to quantify changes in the generated patch [39]. A larger change indicates that the removed section had a greater influence on the model's output [39].

SHAP (SHapley Additive exPlanations) [54] provides a Shapley value for each section that reflects its marginal contribution to the observed output difference. We convert absolute SHAP magnitudes into an attention score by normalizing them within each bug report so that the scores sum to 100 across sections. In practice, this corresponds to an occlusion-style attribution method: if removing section i substantially changes the produced patch, then that section receives higher attention [39]. We then aggregate SHAP-derived attention scores across bugs and compare distributions between successful and unsuccessful repairs. To quantify the magnitude and significance of the differences, we use Cliff's δ effect size measurement and the Mann-Whitney U test ($\alpha = 0.05$), following best practices of statistical reporting [37].

3.1.2 RQ2: How is model attention distributed across specific code and natural-language components within bug report sections? To analyze fine-grained attention patterns within sections, we break down their content into natural language and code components. These components are automatically extracted using regex-based heuristics and manually verified by the first author.

To identify **natural language** components in each section, we perform sentence-level segmentation using the NLTK sentence tokenizer [60]. Each sentence is associated with the Markdown section header under which it appears, yielding the following categories:

- **NL: Description**, sentences describing the overall bug or failure;
- **NL: Version information**, sentences describing system versions or environment configurations;
- **NL: Expected behavior**, sentences describing the intended program behavior;
- **NL: Actual behavior**, sentences describing the observed incorrect behavior;
- **NL: Reproduction**, sentences describing how to reproduce the bug; and
- **NL: Additional information**, sentences describing any supplementary details related to the bug.

To identify **code-related** information, we apply regex (e.g., triple backticks, log formats) tailored to the two languages in our dataset, Python and Java. Using this approach, we derive four categories:

- **Code: Stacktrace** corresponding to the runtime error traces that often indicate the failure location;
- **Code: Test** snippets representing the reproduction tests or assertions used to demonstrate the bug;
- **Code: Class and method** definitions that illustrate relevant portions of the system's implementation; and
- **Code: Import and variable** declarations that reference external APIs or system interfaces.

We manually verify all identified components to ensure accurate classification. Each component is then treated as an independent information unit for evaluating its influence on repair behavior.

For each bug instance, we denote the output obtained by retaining all sections of a bug report as O_{base}. We then perform perturbation analysis within a section by masking one component at a time, keeping the remaining components unchanged. This perturbed bug report is then passed to the model to generate a new patch. To quantify the influence of each component, we measure the semantic difference between the original output (O_{base}) and the perturbed output (O_{pert}) using UniXcoder [23]. UniXcoder provides code-aware embeddings that incorporate syntactic information through abstract syntax tree representations. The importance score of component c is defined as the change in similarity between O_{base} and O_{pert}. Intuitively, if masking a component causes a large semantic change in the generated patch, the removed information is considered to have a strong influence on the model's repair behavior. Aggregating these importance scores across components in different sections of a bug report creates an attention pattern distribution map for each bug report.

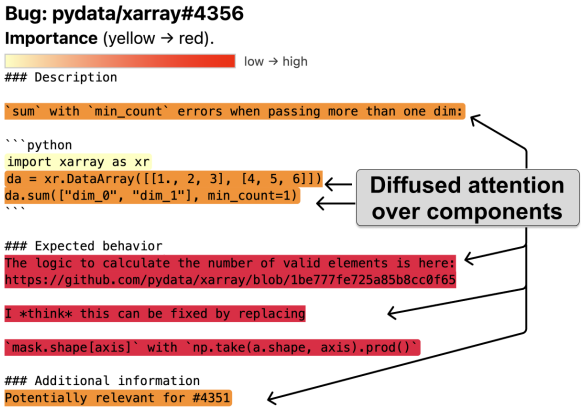

Figure 2: Example of Diffused Attention Pattern in LLMs.

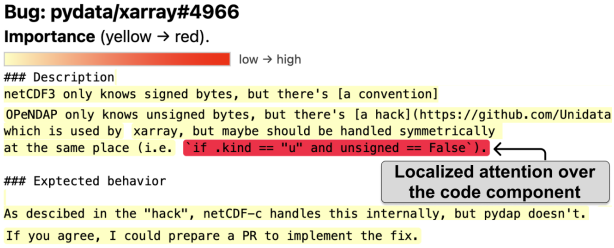

Figure 3: Example of Localized Attention Pattern in LLMs.

We visualize component-level importance scores by mapping them back onto the original bug report. Each component is highlighted using a color scale from yellow to red, with warmer colors indicating greater model attention (see Figure 2). We then analyze these visualizations to identify recurring attention patterns across bugs. Through this process, we identify three attention structures:

- **Diffused**, where model attention is distributed across multiple components (e.g., Figure 2, where attention is distributed across *NL: Expected Behavior*, *Code: Class and method*, *NL: Description*, and *NL: Additional information*);
- **Localized**, where the model focuses heavily on a single component while ignoring other information (e.g., Figure 3, where model attention is only on *Code: Class and method*);
- **No-attention**, where masking any of the components produces no changes in the generated output (e.g., Figure 4).

We also investigate which components (e.g., stacktraces or natural language sections) receive the strongest attention in successful bug repair scenarios. To evaluate the relationship between attention patterns and repair success, we construct contingency tables

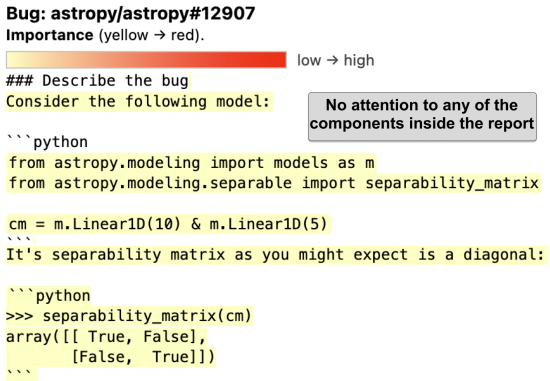

Figure 4: Example of No-attention Pattern in LLMs.

relating the presence of each attention behavior to patch correctness. We apply Fisher's Exact Test due to the binary nature of the patterns [73]. Fisher's test is appropriate for our data because it computes exact probabilities and performs reliably with relatively small sample sizes [35]. To control for multiple hypothesis testing, we apply the Benjamini-Hochberg False Discovery Rate (FDR) correction [4], following recommended statistical reporting in empirical software engineering [37]. In addition, we use logistic regression to report odds ratios and confidence intervals to quantify the magnitude and direction of each attention pattern's association with repair success. To account for bug difficulty as a potential confounding factor, we also repeat the logistic regression analysis by including the benchmark difficulty labels (*Easy*, *Medium*, *Hard*) as control variables, with *Easy* used as the reference category. This allows us to estimate the association between each attention pattern and repair success after adjusting for differences in bug difficulty.

3.1.3 RQ3: How well do model attention patterns align with what developers consider important for bug-fixing? While prior work has shown that the information developers focus on strongly affects their ability to understand and fix bugs [6, 7, 90], existing APR benchmarks do not provide information about which parts of a bug report developers consider most useful during repair. To fill this gap, we construct the first developer attention dataset. From our dataset of 319 bugs, we draw a stratified sample of 100 bug reports (50 Python and 50 Java). This sample size balances sufficient statistical power with approximately 95% confidence and a ±8% margin of error [10], with the substantial manual effort required for detailed attention annotation.

We recruited four experienced software developers (with 5+ years of programming experience in Python and Java) to manually annotate these 100 bug reports. Each bug report is annotated by one developer, each labeling 25 bugs. Using insights from prior studies on how developers identify relevant information from software artifacts [8, 66, 68, 90], we designed the annotation study to capture which bug report content developers consider most important for repair. For each bug report, **(1)** the developers rate the importance of each section on a five-point Likert scale (1 = not important, 5 = very important) [51], **(2)** select the top two most important sections, and **(3)** provide exact-quote key phrases within the selected sections that capture essential information.

Each bug report is annotated by a single experienced developer because the goal of the study is to capture subjective notions of importance during bug repair to test alignment of LLMs and developers rather than establish a single objective ground truth to test alignment between human developers. Different developers may prioritize different information when debugging. However, to reduce excessive subjectivity, before annotation began, we prepared detailed annotation guidelines and conducted a pilot session to refine the procedure and ensure a consistent understanding of the task. In addition, the first author annotated 40 sampled bugs (10 from each annotator) and measured agreement overlap with the original annotations. We see an average Spearman correlation [29] of 0.817 for section-level ratings, a Cohen's Kappa [55] of 0.77 for Top-2 selections, and an average Top-2 overlap of 1.475/2 sections, which all suggest strong consistency across the overlapped annotations. Annotations are collected using a custom Streamlit-based [71] interface developed for this study. More information on annotation instructions and the tool is included in our replication package [61].

We compare these annotations with model attention at two levels of granularity: a) section-level, and b) phrase-level.

a) Section-level: We derive section-level model attention by aggregating the importance scores from *RQ2* within each bug-report section. This yields a normalized attention vector $\{a_s\}$ over sections, where a_s reflects the total importance assigned to information units appearing under section s. We measure alignment using three metrics. First, we compute *Hit@1*, the overlap between the developer-selected top section H_{top1} and the model's top attended section M_{top1} ($|H_{\text{top1}} \cap M_{\text{top1}}|$), and *Hit@2*, the overlap between the developer-selected top-two sections H_{top2} and the model's top-two attended sections M_{top2}, i.e., $|H_{\text{top2}} \cap M_{\text{top2}}|$. Then, we compute the *Spearman rank correlation* between the developer section importance ratings $\{r_s\}$ and the model section attention scores $\{a_s\}$ for each bug report. **b) Phrase-level**: To compare developer-identified key phrases to model attention, we use the results in *RQ2* to obtain a phrase-level attention score over bug reports. We map each developer-provided quote to its corresponding token span, yielding a set of key phrases K. We then measure phrase-level alignment using San Martino's token overlap metrics [39], which quantify the overlap between the set of top-k phrases attended by the model and the developer-annotated key phrases. Specifically, we report Precision@k, Recall@k, and F1@k for k = 10, 20.

4 Results and Discussion

4.1 RQ1: How is model attention allocated across bug report sections in successful and unsuccessful repairs?

Results. As shown in Table 1, *claude-4* achieves the highest fix rate (65%), followed by *gpt-oss* (51%) and *qwen-3* (40%). Across all three models, repair rates are consistently higher for *Easy* bugs (48-71%) than for *Medium* bugs (34-62%). The *Hard* category exhibits 36% to 55% repair rates, but it contains only 11 bugs and does not provide sufficient statistical power for reliable conclusions. A chi-square test of independence with correction [56] finds no statistically significant association between benchmark difficulty and repair success ($p = 0.06$), suggesting that benchmark difficulty alone does not fully explain repair outcomes.

Across our models, successful repairs exhibit high similarity to the ground-truth patches (median CodeBLEU: 91% for *claude-4*,

Table 1: Fix-rate of models across our dataset by difficulty.

Model	Data	Overall	Easy	Medium	Hard
qwen-3-32b	Python	96/248 (38.7%)	59/124 (47.6%)	37/120 (30.8%)	0/4 (0.0%)
	Java	34/71 (47.8%)	10/20 (50.0%)	18/44 (40.9%)	6/7 (85.7%)
	All	**130/319 (40.7%)**	**69/144 (47.9%)**	**55/164 (33.5%)**	**6/11 (54.5%)**
gpt-oss-20b	Python	133/248 (53.6%)	72/124 (58.1%)	60/120 (50.0%)	1/4 (25.0%)
	Java	32/71 (45.1%)	10/20 (50.0%)	19/44 (43.2%)	3/7 (42.9%)
	All	**165/319 (51.7%)**	**82/144 (56.9%)**	**79/164 (48.2%)**	**4/11 (36.4%)**
claude-4-sonnet	Python	165/248 (66.5%)	92/124 (74.2%)	72/120 (60.0%)	1/4 (25.0%)
	Java	43/71 (60.6%)	10/20 (50.0%)	30/44 (68.2%)	3/7 (42.9%)
	All	**208/319 (65.2%)**	**102/144 (70.8%)**	**102/164 (62.2%)**	**4/11 (36.4%)**

85% for *gpt-oss*, and 84% for *qwen-3*), indicating that correct repairs closely match the developer implementations despite syntactic differences such as variable names or code formatting. In contrast, failed repairs show substantially lower similarity to the ground-truth patches (mean CodeBLEU: 30% for *claude-4*, 29% for *gpt-oss*, and 31% for *qwen-3*). This indicates that failures are generally different from the ground-truth implementations rather than near-correct fixes. Our manual inspection showed that failed repairs do not capture the intended bug behavior described in the reports. We discuss representative examples of these failures throughout the discussions of each *RQ*.

Table 2 reports attention differences between solved and unsolved repairs by section for each model using Cliff's δ. Positive values indicate that attention scores tend to be higher for successful repairs, while negative values indicate higher attention for unsuccessful repairs. Across all models, *Bug description* and *Version information* sections exhibit the largest differences. The *Bug description* section shows large positive effect sizes of **0.54, 0.51**, and **0.50** for *claude-4*, *gpt-oss*, and *qwen-3*, respectively, indicating that successful repairs tend to allocate more attention to the description of the problem. In contrast, the *Version information* section shows large negative effect sizes of **-0.53**, **-0.20**, and **-0.42** for each model, suggesting that unsuccessful repairs tend to allocate more attention to environmental metadata. Despite differences in overall repair rates across models, the direction and magnitude of the attention differences are consistent. This suggests that similar attention patterns distinguish successful from unsuccessful repairs across models. These same patterns also appear across both Python and Java bugs, suggesting stability across these programming languages.

Other sections, including *Reproduction*, *Actual Behavior*, *Expected Behavior*, and *Additional Information* show small or inconsistent effect sizes across models, indicating limited differences in attention allocation between successful and unsuccessful repairs.

Discussion. Our findings suggest that successful repairs rely more heavily on the *Bug description* section, which provides the most direct explanation of the failure. Bug descriptions often summarize the underlying issue in natural language and highlight key symptoms of the bug, giving the models clearer signals about what behavior needs to be corrected.

Table 2: Comparing section attention between solved and unsolved bugs using Effect-size (Cliff's δ). Top-2 biggest differences in each model are highlighted (red: top-1, orange: top-2). Bold* denotes statistical significance ($p < 0.05$) with the Mann-Whitney U test.

	claude-4-sonnet			gpt-oss-20b			qwen-3-32b		
Section	**All**	**Python**	**Java**	**All**	**Python**	**Java**	**All**	**Python**	**Java**
Bug description	**0.54***	**0.53***	**0.55***	**0.51***	**0.51***	**0.49***	**0.50***	**0.48***	**0.52***
Version information	**−0.53***	**−0.52***	**−0.48***	−0.20	−0.22	−0.13	**−0.42***	**−0.47***	−0.32
Reproduction	−0.08	−0.12	0.08	0.09	0.15	−0.05	0.14	0.05	0.35
Actual behavior	−0.30	−0.29	−1.00	−0.03	−0.20	0.75	−0.08	−0.02	0.0
Expected behavior	−0.21	−0.25	0.06	0.00	0.05	−0.04	−0.04	−0.13	0.22
Additional information	0.16	0.12	0.34	−0.04	0.04	−0.25	0.18	0.15	0.20

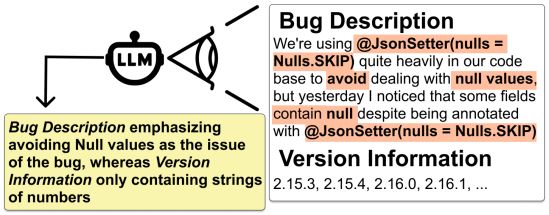

Figure 5: Example of Bug Description and Version Information Sections in Project *jackson-databind#4469*.

In contrast, the *Version information* section often contains environmental metadata such as version numbers or system configurations. While this information can help developers reproduce issues in different versions of the code, it can rarely provide actionable insights into the cause of the bug to LLMs. Figure 5 presents such an example from the *jackson-databind* project. The bug concerns certain fields being returned as null values even though they should have been skipped. The *Bug description* explicitly states this failure condition, directly pointing the core issue. However, the version section only lists the software versions where the issue happens. If the model allocates substantial attention to this metadata, it may focus on strings of numbers that provide little guidance for diagnosing the bug while ignoring other relevant facts. The ground-truth patch, in this bug, simply adds a missing `'if (value == null)'` check. However, *claude-4* fails to generate this condition, showing how emphasizing less informative metadata over the diagnostic description can lead to an incorrect repair.

Another notable observation in our analysis is that *Bug description* and *Version information* are relatively consistent in structure across bug reports. The description typically contains natural language explanations of the failure, while version sections contain environment details. Other sections are far more heterogeneous in terms of content. For example, *Reproduction* or *Actual Behavior* may contain code snippets, stacktraces, or natural language explanations, which vary substantially across reports. Since the information in these sections differs substantially from one report to another, their section-level attention signals are likely diluted, resulting in smaller and less consistent effect sizes. Based on these observations, it is possible that attention differences may be driven

Table 3: Associations between attention patterns and repair success using Fisher's Exact Test with Benjamini-Hochberg FDR correction (FDR q). Odds Ratios (OR) are reported with bootstrap Confidence Intervals (CI). The column *Influence* indicates the direction of the association based on the OR value (OR > 1 more in successful; OR < 1 more in unsuccessful repairs). Rows with statistically significant associations ($q < 0.05$) are marked with * and shaded according to the direction of their effect: green indicates a positive association with successful repairs, while red indicates a negative association with repair success.

		Overall				Python				Java			
Pattern	**Influence**	**OR [CI]**	***p*-value**	**FDR q**	**Count**	**OR [CI]**	***p*-value**	**FDR q**	**Count**	**OR [CI]**	***p*-value**	**FDR q**	**Count**
Attention Structure													
No-attention	·	1.0 [1.0,1.0]	8.87×10^{-1}	9.60×10^{-1}	64	1.0 [1.0,1.0]	1.97×10^{-1}	2.56×10^{-1}	51	1.0 [1.0,1.0]	6.65×10^{-2}	1.44×10^{-1}	13
Diffused*	▲	**2.07 [1.68,2.52]**	3.71×10^{-15}	2.41×10^{-14}	117	1.66 [1.28,2.13]	4.63×10^{-9}	3.01×10^{-8}	84	1.86 [1.53,2.21]	7.99×10^{-8}	1.00×10^{-6}	33
Localized*	▼	**0.40 [0.33,0.50]**	7.38×10^{-16}	9.60×10^{-15}	138	0.40 [0.32,0.53]	9.86×10^{-12}	1.28×10^{-10}	113	0.65 [0.54,0.81]	7.25×10^{-5}	4.71×10^{-4}	25
Focus Targets													
NL:Description*	▲	1.42 [1.09,1.85]	1.05×10^{-2}	2.79×10^{-2}	129	1.12 [0.85,1.48]	4.22×10^{-1}	4.57×10^{-1}	95	1.57 [1.24,1.93]	3.37×10^{-4}	1.46×10^{-3}	34
NL: Expected behavior	▲	1.31 [1.06,1.65]	6.37×10^{-2}	1.04×10^{-1}	21	1.30 [1.05,1.63]	4.83×10^{-2}	1.57×10^{-1}	18	1.04 [0.94,1.15]	6.04×10^{-1}	7.14×10^{-1}	3
NL: Reproduction	▲	1.23 [0.96,1.59]	3.04×10^{-2}	5.65×10^{-2}	24	1.15 [0.88,1.49]	1.51×10^{-1}	2.56×10^{-1}	20	1.13 [1.03,1.26]	4.77×10^{-2}	1.24×10^{-1}	4
NL: Actual behavior	▲	1.06 [0.90,1.24]	1.26×10^{-1}	1.63×10^{-1}	7	1.10 [0.93,1.28]	1.12×10^{-1}	2.56×10^{-1}	7	1.00 [1.00,1.00]	1.00	1.00	0
NL: Additional information	▼	0.98 [0.78,1.24]	1.00	1.00	12	0.94 [0.78,1.11]	1.00	1.00	9	1.05 [0.89,1.24]	6.04×10^{-1}	7.14×10^{-1}	3
NL:Version information*	▼	0.61 [0.49,0.77]	1.41×10^{-2}	3.05×10^{-2}	27	0.72 [0.58,0.89]	1.66×10^{-1}	2.56×10^{-1}	21	0.81 [0.69,0.93]	2.56×10^{-2}	8.33×10^{-2}	6
Code:Stacktrace*	▲	1.41 [1.11,1.81]	3.61×10^{-3}	1.57×10^{-2}	54	1.36 [1.06,1.74]	5.17×10^{-3}	2.24×10^{-2}	41	1.12 [0.94,1.33]	3.62×10^{-1}	5.22×10^{-1}	13
Code: Classes and methods	▲	1.26 [0.94,1.69]	7.26×10^{-2}	1.05×10^{-1}	45	1.19 [0.88,1.61]	2.60×10^{-1}	3.08×10^{-1}	35	1.14 [0.94,1.35]	1.78×10^{-1}	2.89×10^{-1}	10
Code:Test*	▲	1.16 [1.03,1.33]	1.07×10^{-2}	2.79×10^{-2}	5	1.07 [1.00,1.19]	1.48×10^{-1}	2.56×10^{-1}	2	1.11 [1.00,1.24]	1.05×10^{-1}	1.94×10^{-1}	3
Code: Imports and variables	▲	1.16 [0.89,1.53]	1.81×10^{-1}	2.14×10^{-1}	43	1.15 [0.88,1.50]	1.84×10^{-1}	2.56×10^{-1}	34	1.06 [0.91,1.25]	7.29×10^{-1}	7.89×10^{-1}	9

by finer-grained components within the sections rather than the sections themselves. We examine this more closely in *RQ2*.

Across models, successful repairs consistently allocate most attention to the *Bug description*, whereas unsuccessful repairs allocate more attention to *Version information*. Other sections show weaker or no consistent differences.

4.2 RQ2: How is model attention distributed across specific code and natural-language components within bug report sections?

Results. Table 3 summarizes the *RQ2* results by quantifying how different attention patterns over bug report components are associated with repair success, both overall and by programming languages (Python and Java). For each pattern, we report Fisher's exact test results (p-value) with Benjamini-Hochberg FDR correction (FDR q), logistic regression odds ratios with confidence intervals (OR [CI]), and the number of bugs in which the pattern appears (Count). The *Influence* column indicates the direction of the association: $OR > 1$ denotes a positive association with successful repairs, whereas $OR < 1$ denotes a negative association.

We observe strong differences in overall repair success depending on how attention is distributed across bug report components. ***Diffused attention***, where the model allocates attention across multiple components, is strongly associated with successful repairs (OR = 2.07, $q = 2.41 \times 10^{-14}$). In contrast, ***Localized attention***, where the model concentrates on a specific part of the report, is negatively associated with repair success (OR = 0.40, $q = 9.60 \times 10^{-15}$). These results suggest that successful repairs typically require integrating multiple sources of information from the bug report rather than focusing narrowly on a single component. When ***no-attention*** is observed, we find no association with repair success or failure.

This pattern is consistent across both Python and Java. Diffused attention remains positively associated with success in Python (OR = 1.66, $q = 3.01 \times 10^{-8}$) and Java (OR = 1.86, $q = 1.00 \times 10^{-6}$), while localized attention remains negatively associated in both Python (OR = 0.40, $q = 1.28 \times 10^{-10}$) and Java (OR = 0.65, $q = 4.71 \times 10^{-4}$). Although the negative association for localized attention is weaker in Java, the overall direction remains the same, suggesting that the relationship between attention structure and repair outcome is stable across programming languages.

Among components, code-related information within bug reports plays an important role in successful repair. In particular, attention to *Code: Stacktrace* shows a significant positive association with repair success (OR = 1.41, $q = 1.57 \times 10^{-2}$). This result is consistent with the diagnostic role of stacktraces in identifying potential failure locations [68]. Similarly, attention to *Code: Test* is positively associated with success (OR = 1.16, $q = 2.79 \times 10^{-2}$), indicating that models benefit from examples that concretely demonstrate the failing behavior. However, the relatively small count for test code (Count = 5) suggests that this result should be interpreted more cautiously than higher-count components such as stacktraces (Count = 54). Other code components, such as *Code: Classes and methods* definitions and API-related usage (*Code: Imports and variables*), show positive but statistically non-significant associations with repair success. While these elements may provide useful implementation context, their influence appears less consistent across bugs when compared to stacktrace or test information.

Among natural language components, attention to the *NL: Description* shows a significant positive association with successful repair (OR = 1.42, $q = 2.79 \times 10^{-2}$), showing that models benefit

Table 4: Associations between attention patterns and repair success while controlling for bug difficulty (*Easy*: Baseline). Odds Ratios (OR) are reported with Confidence Intervals (CI).

Pattern	Influence	Coef.	OR [CI]
Attention Structure			
Diffused	▲	0.692	**2.00 [1.54, 2.56]**
Localized	▼	-0.962	**0.38 [0.30, 0.49]**
Focus Targets			
NL:Description	▲	0.313	1.37 [1.05, 1.81]
NL:Expected behavior	▲	0.260	1.30 [1.05, 1.61]
NL:Reproduction	▲	0.202	1.22 [0.95, 1.57]
NL:Actual behavior	▲	0.094	1.10 [0.94, 1.27]
NL:Additional information	▼	-0.023	0.98 [0.77, 1.24]
NL:Version information	▼	-0.468	0.63 [0.50, 0.79]
Code:Stacktrace	▲	0.338	1.40 [1.08, 1.78]
Code:Classes and methods	▲	0.226	1.25 [0.93, 1.66]
Code:Test	▲	0.139	1.15 [1.03, 1.29]
Code:Imports and variables	▲	0.146	1.16 [0.88, 1.51]
Bug Difficulty (Control Variables)			
Difficulty:Hard	▼	-0.037	0.96 [0.78, 1.19]
Difficulty:Medium	▼	-0.426	0.65 [0.50, 0.85]

from detailed problem descriptions. In contrast, attention to *NL: Version information* is negatively associated with success (OR = 0.61, $q = 3.05 \times 10^{-2}$), suggesting that overemphasizing only on environmental metadata could be less relevant to produce correct patches. This is consistent with our *RQ1* results, where successful repairs allocate more attention to the *Bug description* section and less to *Version information*, both of which mainly consist of natural language components. Other natural language components, such as *expected behavior*, *reproduction steps*, and *actual behavior*, show positive trends but do not remain statistically significant after multiple testing correction. The *additional information* section shows a weak negative trend, likely because it frequently contains external links or references that are inaccessible to the model.

Table 4 reports the logistic regression results after controlling for bug difficulty. Overall, our findings remain highly consistent after accounting for difficulty. Diffused attention continues to show a strong positive association with repair success (OR = 2.00), while localized attention remains strongly negatively associated (OR = 0.38). Attention to *NL: Description* (OR = 1.37), *Code: Stacktrace* (OR = 1.40), and *Code: Test* (OR = 1.15) remains positively associated with successful repair, whereas attention to *NL: Version information* continues to show a negative association (OR = 0.63). These effect sizes are nearly identical to those observed in Table 3 (e.g., Diffused attention 2.07 vs. 2.00), indicating that the identified patterns are robust even after accounting for difficulty. Although *Medium* and *Hard* bugs are associated with lower repair success than *Easy* bugs, the consistency of the coefficients shows that bug difficulty alone does not explain the observed behaviors.

Discussion. Diffused attention shows to be the most effective attention pattern for repair across bugs with different difficulties because the model integrates multiple components within the bug report instead of relying on a single source of information. Figure 6 shows an example from project *mockito*. In this case, the bug occurs from incorrect abstraction of enums, and the correct repair requires understanding both the failing test and the expected behavior. By distributing attention across the test information describing the current and expected behavior, the model is able to generate the correct patch, matching the ground-truth implementation. A similar pattern appears in Figure 2, which shows a bug report from project *xarray*. Here, the model attends to both natural language explanations and code components describing the intended logic. By combining these complementary signals, the model is able to correctly modify the function to accept more than one dimension, consistent with the ground-truth patch.

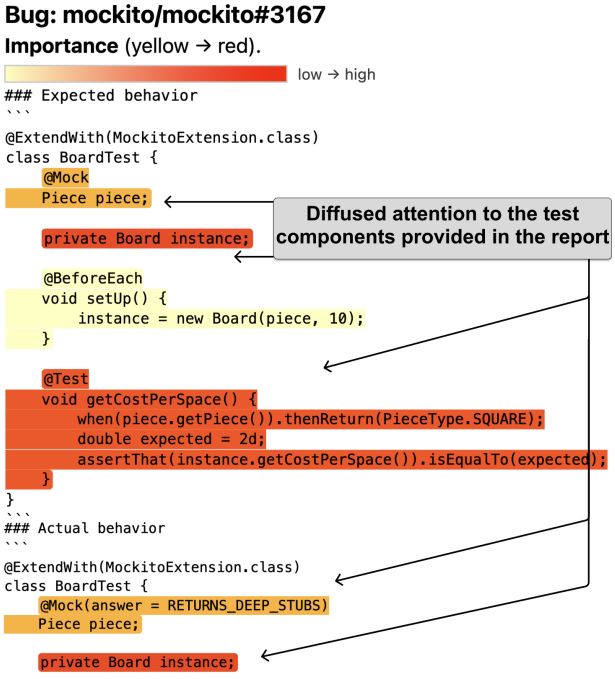

Figure 6: Example of Diffused Attention Throughout the Report in Project *mockito*.

Localized attention, however, can act as a double-edged sword. Figure 3 shows a bug from project *xarray* where focusing on a specific developer-provided hint, the model generates the correct patch. However, this behavior often leads to failure when the localized components are misleading or incomplete. For example, Figure 7 presents two bugs from the project *django*. In the first case, the model fixates on a sentence referencing an external pull request that it cannot access while overlooking the earlier description explaining that the bug is caused by a `'TypeError'`. The ground-truth patch simply adds a missing `'TypeError'` check, but the generated patch misses this condition and therefore fails. In the second case, the model concentrates on reference links in the additional information section while ignoring diagnostic system logs earlier in the report. The ground-truth patch updates the `'autoreloader'` to correctly pass the `'-X'` options described in these logs, whereas the generated patch fails to incorporate this behavior. These examples show that while localized attention may occasionally succeed when the focused component directly encodes the solution, successful repairs more consistently emerge when models distribute attention across multiple complementary components within the bug report.

In addition, to assess whether these observations are specific to *qwen3-32b*, we repeated our *RQ2* analysis on the Java subset (71

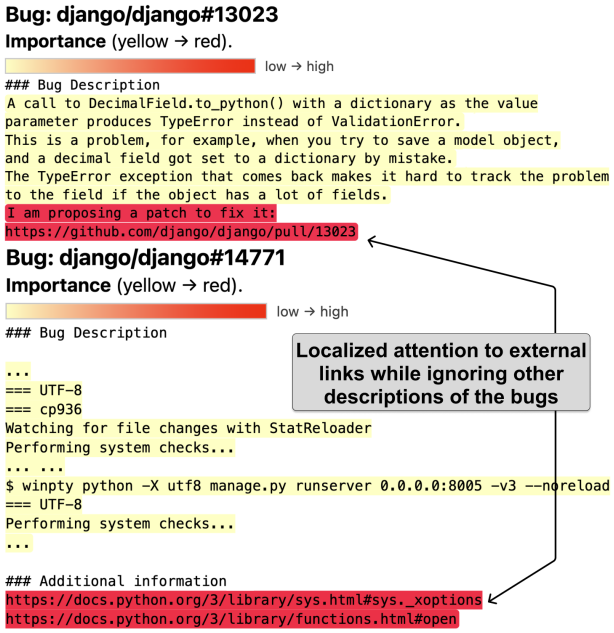

Figure 7: Two Examples of Localized Attention in Reports of *django* Project.

bugs) using *gpt-oss-20b*. As shown in Table 5, we observe the same overall attention patterns. Diffused attention remains positively associated with repair success (OR = 1.56), while localized attention is negatively associated (OR = 0.54), and diagnostically relevant components such as stacktraces (OR = 1.24) and natural-language descriptions (OR = 1.19) again show positive associations with successful repair. These consistent trends across different LLMs suggest that the broader attention behaviors identified in our study are not unique to a single model.

Table 5: Associations between attention patterns and repair success for *gpt-oss-20b* on Java subset. Bold* denotes statistical significance ($p < 0.05$).

Pattern	Influence	OR [CI]	p-value	Count
Attention Structure				
No-attention	·	1.0 [1.0, 1.0]	4.54×10^{-1}	8
Diffused*	▲	**1.56 [1.26, 1.90]**	$\mathbf{1.34\times10^{-5}}$	35
Localized*	▼	**0.54 [0.45, 0.66]**	$\mathbf{1.09\times10^{-7}}$	28
Focus Targets				
NL:Description	▲	1.19 [0.94, 1.51]	9.16×10^{-2}	40
NL:Expected behavior	▲	1.13 [0.96, 1.32]	8.39×10^{-2}	6
NL:Reproduction	▲	1.06 [0.88, 1.28]	3.30×10^{-1}	10
NL:Actual behavior	▼	0.95 [0.84, 1.00]	1.00	1
NL:Additional information	▲	1.06 [0.92, 1.23]	6.93×10^{-1}	7
NL:Version information	▼	0.92 [0.78, 1.09]	6.83×10^{-1}	6
Code:Stacktrace*	▲	**1.24 [1.02, 1.49]**	$\mathbf{1.77\times10^{-2}}$	11
Code:Test	▲	1.15 [0.96, 1.35]	8.39×10^{-2}	6
Code:Imports and variables	▲	1.07 [0.90, 1.27]	7.22×10^{-1}	9
Code:Classes and methods	▼	0.90 [0.72, 1.12]	1.00	21

Successful repairs are characterized by *diffused attention* across multiple components, with strong focus on diagnostic information such as bug descriptions and stacktraces. In contrast, failures often involve more *localized attention*, especially on non-diagnostic metadata.

4.3 RQ3: How well do model attention patterns align with what developers consider important for bug-fixing?

Results. Table 6 reports how frequently each bug report section appears among the top-2 most important sections identified by developers and by the LLM. Overall, both developers and the LLM most frequently prioritize the *Bug description* section, appearing in 79% and 61% of bugs, respectively, with a substantial overlap of 54%. The *Reproduction* section is the second most commonly prioritized (43% for developers and 34% for the LLM), showing a high overlap (30%). Other sections are selected less frequently. Developers emphasize *Expected behavior* more often (18%) than the LLM (11%), while the LLM slightly prioritizes *Version information* and *Actual behavior* more than developers.

Table 6: Frequency and Overlap of the top-2 most important sections selected by developers vs. LLM.

Section	Developer Top-2 (%)	LLM Top-2 (%)	Overlap (%)
Bug description	79 (79%)	61 (61%)	54 (54%)
Reproduction	43 (43%)	34 (34%)	30 (30%)
Expected behavior	18 (18%)	11 (11%)	6 (6%)
Additional information	16 (16%)	15 (15%)	8 (8%)
Version information	6 (6%)	11 (11%)	5 (5%)
Actual behavior	4 (4%)	7 (7%)	2 (2%)

Table 7 reports the mean alignment scores of successful and unsuccessful repairs, along with the Mann-Whitney U test p-values. At the section level, successful repairs exhibit significantly stronger alignment with developer importance ratings. The average Spearman correlation between developer Likert ratings and model attention is significantly higher for successful repairs (rho = 0.60) compared to unsuccessful ones (rho = 0.34), yielding a statistically significant difference (p = 0.036). This suggests that when models prioritize bug report sections similarly to developers, they are more likely to generate correct patches. Hit@2 and Hit@1, which capture the overlap between the sections that developers and model rated highest, show that successful repairs achieve a higher average score (1.17 vs. 0.93, 0.57 vs. 0.33, respectively), with Hit@1 being statistically significant (p = 0.02).

Table 7: Alignment metrics for successful vs. unsuccessful Repairs. Bold* denotes statistical significance ($p < 0.05$) with the Mann-Whitney U test. Column Δ shows the mean difference (Success-Fail).

Level	Alignment Metric	Success	Fail	Δ	p-value
Section	**Spearman ρ***	0.60	0.34	0.26▲	**0.036**
	Hit@2	1.17	0.93	0.24▲	0.105
	Hit@1*	0.57	0.33	0.24▲	**0.028**
Phrase	**F1@20***	0.19	0.08	0.11▲	**0.031**
	Precision@20*	0.42	0.18	0.24▲	**0.027**
	Recall@20*	0.13	0.05	0.08▲	**0.048**
	F1@10*	0.10	0.04	0.06▲	**0.019**
	Precision@10*	0.37	0.16	0.21▲	**0.013**
	Recall@10*	0.06	0.03	0.03▲	**0.023**

A similar trend appears at the phrase level as well. Successful repairs consistently achieve higher overlap between model-attended phrases and developer key phrases across all metrics. Successful repairs show higher F1@20 (0.19 vs. 0.08, $p = 0.031$), Precision@20 (0.42 vs. 0.18, $p = 0.027$), and Recall@20 (0.13 vs. 0.05, $p = 0.048$) (same trend for k=10). These results indicate that models producing correct repairs are more likely to focus on the specific phrases that developers identify as important for diagnosing the bug.

Discussion. Prior work has shown that during bug fixing, developers rely heavily on specific bug report elements, such as bug description, test cases, and stack traces [6, 7, 90]. Our findings extend this perspective to LLM-based program repair; LLMs are more likely to succeed when model attention aligns with the sections and phrases that developers consider most important.

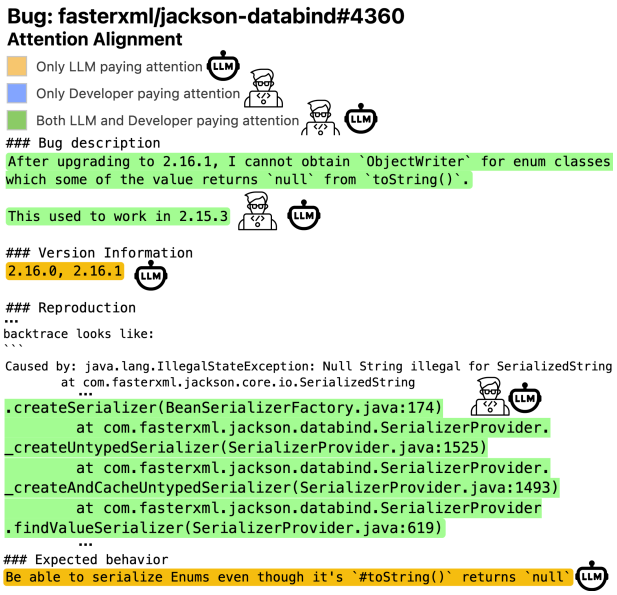

Figure 8: Aligned attention between developer and LLM for a bug in project *jackson-databind*.

Figure 8 shows an example of aligned attention in a bug report from project *jackson-databind*. In this case, both the developer and model concentrate on the key descriptions of the bug and stacktrace. These components clearly reveal the underlying issue, which is the incorrect handling of enum serialization. And relying on this information, the model successfully generates a patch matching the ground-truth implementation. However, Figure 9 shows an example from project *pytest* where there is a clear attention divergence between model and developer. The developer focuses on the description and test information that reveal the failing behavior. The model pays attention to version information the most. While these components provide environmental context, they offer limited value for identifying the root cause of the bug. The ground-truth patch modifies the function so that tests are no longer executed when the class has been skipped using '`unittest.skip`' or '`pytest.mark.skip`'. However, the model fails to generate this behavior, resulting in an incorrect repair.

Our finding adds an important task-specific perspective to prior work on human-model attention alignment [48, 62]. Prior work in code generation has shown that human-model misalignment can help explain model errors [39, 46, 59]. Our results extend this insight to program repair, suggesting that incorrect patches may arise when model attention diverges from the diagnostic evidence that developers consider important. Studying such alignment, therefore, makes repair behavior more interpretable. In future, our developer attention dataset could also serve as useful supervision for guiding the fine-tuning of APR models.

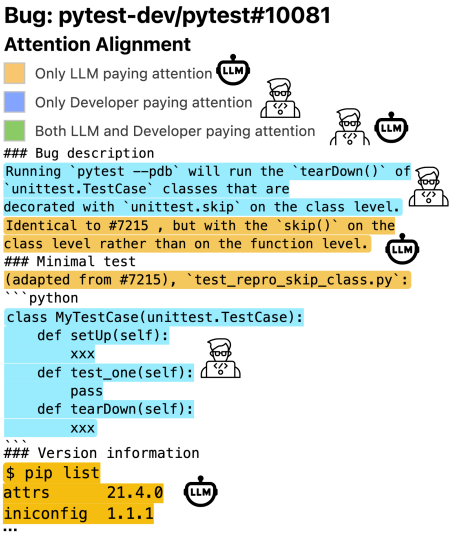

Figure 9: Misaligned attention between developer and LLM for a bug in project *pytest*.

Stronger attention alignment between model and developers, both at the section and phrase levels, is associated with higher repair success.

5 Related Work

LLM-based Program Repair. Prior work has explored a wide range of techniques for improving the effectiveness of LLMs for APR. Early work showed that even static prompting with LLMs can outperform traditional APR tools [78], and subsequent systems such as MMAPR [83], RING [33], and InferFix [32] further improved repair performance using methods such as few-shot prompting. Benchmarks built from real-world software bugs, such as *SWE-bench* [31], highlighted the challenges of applying LLMs to realistic bug repair tasks. However, studies showed that incorporating richer contextual signals into LLMs can significantly improve their performance. Fault localization signals [21], failing tests [79], relevant code [30, 76], and stacktraces [24, 34] can all improve repair accuracy. Historical information, such as prior commits, has also been shown to provide useful context for patch generation [67]. Parasaram et al. proposed MANIPLE, which includes relevant bug-related facts from a repository into prompts to improve patch generation [63]. Ehsani et al. introduced a hierarchical knowledge injection technique that adds contextual information to LLMs in three layers (bug, repository, project) at a time, achieving significant improvements over MANIPLE [12]. More recently, agentic systems such as SWE-Agent [81], OpenHands [20], AutoCodeRover [87],

and iSWE-agent [22] combine multi-round reasoning, repository exploration, and tool interactions to iteratively generate and validate patches. At the time of writing, Sonar Foundation Agent [69] and live-SWE agent [77] both achieve the highest repair performance on *SWE-bench Verified*, resolving up to 79% of bugs.

These works highlight the importance of contextual signals for LLM-based bug repair. However, it is unclear how LLMs interpret the information they receive. To our knowledge, our work is the first to analyze how LLMs attend to different information in a bug report, and how these attention patterns relate to repair success.

Model Attention Analysis. Attention in transformer models and its role in interpretability have been extensively examined in previous work. Early studies analyze attention weights across layers and heads to understand how semantic information is encoded [9, 40]. Other works propose alternative analysis techniques, including perturbation-based methods [65], gradient-based attribution [3, 28], and attention aggregation strategies [9, 40], to characterize what attention represents and how it relates to model behavior. More recent efforts extend these ideas to large language models to improve interpretability [11, 88, 89], reduce hallucination [27], and improve inference efficiency through attention optimization [47]. Recent work has also examined attention in LLM-based code generation. Kou et al. [39] show that models often attend to different parts of task descriptions than humans during code generation, highlighting the need for human-aligned LLMs for better interpretability and developer trust. Other works also observed misalignment between model attention and human reasoning, suggesting that attention patterns can reveal failure modes in generated code [46, 59, 62]. Several studies explore methods to align model attention with human signals, such as using eye-tracking data for fine-tuning [86] or structure-aware attention mechanisms to guide models toward meaningful input regions [50]. Overall, these works suggest that attention alignment can not only improve interpretability, but also lead to better task performance [48, 62].

Despite these advances, interpretability remains a major challenge for integrating LLMs reliably into developer workflows [41, 45]. We aim to address this gap by analyzing how LLMs allocate attention over bug reports during repair, and how that compares to information developers consider important for bug fixing.

6 Threats to Validity

Construct Validity. We measure model attention using perturbation analysis by removing parts of the bug report and observing how the model's output changes. While this provides a causal signal (changes in input lead to changes in output), it is still an approximation and may not fully reflect model reasoning. However, prior work shows that perturbation-based methods provide a more direct and human-aligned interpretation compared to other attention analysis methods [11, 28, 39]. For *RQ3*, we use developer judgments to identify important sections and phrases in bug reports. To reduce subjectivity in these manual annotations, we developed annotation guidelines through multiple discussions, recruited experienced developers, conducted a pilot study, and assigned an equal number of bugs to each annotator. We also computed inter-rater agreement on a subset of annotations, showing strong agreement (Spearman = 0.81, Cohen's Kappa = 0.77).

Internal Validity. We use deterministic decoding (zero temperature) and fixed prompts to reduce randomness and isolate the effect of input perturbations. Although attention may vary under stochastic decoding, this controlled setting supports more reliable analysis. Because perturbation removes parts of a bug report, masking may disrupt information flow; we mitigate this by applying the same masking procedure across distinct sections. Repair outcomes may also be influenced by factors such as bug difficulty. To mitigate this threat, in *RQ2*, we repeat our analysis while controlling for bug difficulty, which results in highly identical results. In *RQ3*, each bug report is annotated by a single developer because detailed manual annotation is costly and our goal is to capture subjective notions of importance that can naturally vary across developers.

External Validity. Our dataset includes 319 Python and Java bugs from *SWE-Bench Verified* and *Multi-SWE-Bench*. While our findings might not generalize to other languages, domains, or less-structured bug reports, these widely used APR benchmarks consist of real-world software bugs and provide sufficiently detailed reports for controlled attention analysis. In *RQ1*, we evaluate three LLMs (one proprietary and two open-source), and for a deeper analysis in *RQ2* and *RQ3*, we focus on one LLM due to the substantially higher computational cost of perturbation-based analysis. While these models represent different sizes and types, our findings may not generalize to other LLMs or agentic APR systems. To mitigate this threat, we first compare section-level attention across all three models in *RQ1*, observing consistent trends despite differences in repair performance. We further validate the generality of the fine-grained analysis by repeating *RQ2* on a Java subset (71 bugs) using *gpt-oss-20b*, obtaining the same overall attention patterns as *qwen3-32b*. In *RQ3*, our analysis focuses on a subset of 100 manually annotated bug reports, which might not generalize to larger datasets. However, for our dataset, this sample size is statistically significant, providing approximately 95% confidence with a margin of error of ±8% [10].

7 Conclusion and Future Work

Using perturbation-based analysis over 319 real-world bugs, we examined how models allocate attention across bug reports, how these patterns differ between successful and unsuccessful repairs, and how model attention aligns with what information developers consider important for bug fixing. Our findings show that repair success is strongly associated with how models prioritize information within bug reports. Successful repairs exhibit diffused attention over multiple fine-grained information in diagnostically important components, while failures often arise from over-localized attention toward less informative components. Stronger alignment between model attention and developers is linked to higher repair success, further highlighting attention misallocation as a key factor for failures in LLM-based program repair. While bug difficulty and other factors could influence repair performance, our results show that they do not fully explain repair outcomes. Even after controlling for bug difficulty, the same attention patterns remain strongly associated with successful repairs, which shows the importance of understanding how LLMs process information during repair.

By revealing which parts of bug reports models prioritize during repair, our approach can help practitioners redesign prompts,

retrieval strategies, and pre-processing pipelines to emphasize diagnostically useful information while reordering or reformulating components that consistently distract models or get ignored. Beyond prompt design, these insights can support attention-aware context selection and monitoring mechanisms that identify potential attention misallocation before incorrect repairs are generated. Our annotated dataset can also support future fine-tuning approaches that align LLM information prioritization with developer reasoning. Finally, because our perturbation-based method is model-agnostic and does not require access to internal model states, it can be applied to agentic APR systems to improve their interpretability and better understand how information is processed through complex repair workflows.

Data Availability Statement

Our replication package is available online at this DOI link: https://doi.org/10.5281/zenodo.21381449

References

[1] Nikta Akbarpour, Mahdieh Sadat Benis, Fatemeh Hendijani Fard, Ali Ouni, and Mohamed Aymen Saied. 2025. Collaborative Agents for Automated Program Repair in Ruby. arXiv:2511.03925 [cs.SE]

[2] Anthropic. 2025. https://www.anthropic.com/news/claude-4

[3] Jasmijn Bastings and Katja Filippova. 2020. The elephant in the interpretability room: Why use attention as explanation when we have saliency methods? arXiv:2010.05607 [cs.CL] https://arxiv.org/abs/2010.05607

[4] Yoav Benjamini and Yosef Hochberg. 1995. Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing. *Journal of the Royal Statistical Society Series B: Statistical Methodology* 57, 1 (Jan. 1995), 289–300.

[5] Sicong Cao, Xiaobing Sun, Ratnadira Widyasari, David Lo, Xiaoxue Wu, Lili Bo, Jiale Zhang, Bin Li, Wei Liu, Di Wu, and Yixin Chen. 2025. A Systematic Literature Review on Explainability for ML/DL-based Software Engineering. *ACM Comput. Surv.* 58, 4, Article 95 (Oct. 2025), 34 pages. doi:10.1145/3763230

[6] Oscar Chaparro, Carlos Bernal-Cárdenas, Jing Lu, Kevin Moran, Andrian Marcus, Massimiliano Di Penta, Denys Poshyvanyk, and Vincent Ng. 2019. Assessing the quality of the steps to reproduce in bug reports. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 86–96.

[7] Oscar Chaparro, Jing Lu, Fiorella Zampetti, Laura Moreno, Massimiliano Di Penta, Andrian Marcus, Gabriele Bavota, and Vincent Ng. 2017. Detecting missing information in bug descriptions. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 396–407.

[8] Preetha Chatterjee, Minji Kong, and Lori Pollock. 2020. Finding Help with Programming Errors: An Exploratory Study of Novice Software Engineers' Focus in Stack Overflow Posts. *Journal of Systems and Software* 159 (2020), 110454.

[9] Kevin Clark, Urvashi Khandelwal, Omer Levy, and Christopher D. Manning. 2019. What Does BERT Look at? An Analysis of BERT's Attention. In *Proceedings of the 2019 ACL Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, Tal Linzen, Grzegorz Chrupała, Yonatan Belinkov, and Dieuwke Hupkes (Eds.). Association for Computational Linguistics, Florence, Italy, 276–286.

[10] William G. Cochran. 1977. *Sampling Techniques* (3rd ed.). John Wiley & Sons, New York, NY.

[11] Ximing Dong, Shaowei Wang, Dayi Lin, Gopi Krishnan Rajbahadur, and Ahmed E. Hassan. 2025. Promptexp: Multi-Granularity Prompt Explanation of Large Language Models. In *2025 2nd IEEE/ACM International Conference on AI-powered Software (AIware)*. 01–10. doi:10.1109/AIware69974.2025.00027

[12] Ramtin Ehsani, Esteban Parra, Sonia Haiduc, and Preetha Chatterjee. 2025. Hierarchical Knowledge Injection for Improving LLM-based Program Repair. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1440–1452. doi:10.1109/ASE63991.2025.00122

[13] Ramtin Ehsani, Sakshi Pathak, and Preetha Chatterjee. 2025. Towards Detecting Prompt Knowledge Gaps for Improved LLM-guided Issue Resolution. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*. 699–711. doi:10.1109/MSR66628.2025.00107

[14] Ramtin Ehsani, Sakshi Pathak, Esteban Parra, Sonia Haiduc, and Preetha Chatterjee. 2025. What characteristics make ChatGPT effective for software issue resolution? An empirical study of task, project, and conversational signals in GitHub issues. *Empirical Software Engineering* 31, 1 (Nov. 2025). doi:10.1007/s10664-025-10745-8

[15] An Yang et al. 2025. Qwen3 Technical Report. arXiv:2505.09388 [cs.CL]

[16] Daoguang Zan et al. 2025. Multi-SWE-bench: A Multilingual Benchmark for Issue Resolving. arXiv:2504.02605 [cs.SE]

[17] Md Tahmid Rahman Laskar et al. 2024. A Systematic Survey and Critical Review on Evaluating Large Language Models: Challenges, Limitations, and Recommendations. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (Eds.). Association for Computational Linguistics, Miami, Florida, USA, 13785–13816.

[18] OpenAI et al. 2025. gpt-oss-120b & gpt-oss-20b Model Card. arXiv:2508.10925 [cs.CL] https://arxiv.org/abs/2508.10925

[19] Stella Biderman et al. 2024. Lessons from the Trenches on Reproducible Evaluation of Language Models. arXiv:2405.14782 [cs.CL]

[20] Xingyao Wang et al. 2025. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. arXiv:2407.16741 [cs.SE]

[21] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated Repair of Programs from Large Language Models. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1469–1481. doi:10.1109/ICSE48619.2023.00128

[22] Jatin Ganhotra, Sami Serhan, Antonio Abu Nassar, Avraham Shinnar, Ziv Nevo, and Martin Hirzel. 2026. Resolving Java Code Repository Issues with iSWE Agent. arXiv:2603.11356 [cs.SE] https://arxiv.org/abs/2603.11356

[23] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. arXiv:2203.03850 [cs.CL] https://arxiv.org/abs/2203.03850

[24] Mirazul Haque, Petr Babkin, Farima Farmahinifarahani, and Manuela Veloso. 2025. 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*. Association for Computational Linguistics, Albuquerque, New Mexico, USA, 160–179.

[25] Andreas Hochlehnert, Hardik Bhatnagar, Vishaal Udandarao, Samuel Albanie, Ameya Prabhu, and Matthias Bethge. 2025. A Sober Look at Progress in Language Model Reasoning: Pitfalls and Paths to Reproducibility. arXiv:2504.07086 [cs.LG]

[26] Haichuan Hu, Ye Shang, Weifeng Sun, and Quanjun Zhang. 2025. TSAPR: A Tree Search Framework For Automated Program Repair. arXiv:2507.01827 [cs.SE]

[27] Yuheng Huang, Lei Ma, Keizaburo Nishikino, and Takumi Akazaki. 2025. Risk assessment framework for code llms via leveraging internal states. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*. 432–443.

[28] Sarthak Jain and Byron C. Wallace. 2019. Attention is not Explanation. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, Minneapolis, Minnesota, 3543–3556.

[29] Joseph James. 2026. Counting on Consensus: Selecting the Right Inter-annotator Agreement Metric for NLP Annotation and Evaluation. arXiv:2603.06865 [cs.CL]

[30] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1430–1442.

[31] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=VTF8yNQM66

[32] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. 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* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1646–1656.

[33] Harshit Joshi, José Cambronero Sanchez, Sumit Gulwani, Vu Le, Ivan Radiček, and Gust Verbruggen. 2023. 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 (AAAI'23/IAAI'23/EAAI'23)*. AAAI Press, Article 573, 10 pages.

[34] Jan Keller and Jan Nowakowski. 2024. *AI-powered patching: the future of automated vulnerability fixes.* Technical Report.

[35] Hae-Young Kim. 2017. Statistical notes for clinical researchers: Chi-squared test and Fisher's exact test. *Restorative Dentistry & Endodontics* 42, 2 (May 2017), 152–155. doi:10.5395/rde.2017.42.2.152

[36] Shunsuke Kitada and Hitoshi Iyatomi. 2021. Attention Meets Perturbations: Robust and Interpretable Attention With Adversarial Training. *IEEE Access* 9 (2021), 92974–92985. doi:10.1109/ACCESS.2021.3093456

[37] Barbara Kitchenham, Lech Madeyski, David Budgen, Jacky Keung, Pearl Brereton, Stuart Charters, Shirley Gibbs, and Amnart Pohthong. 2017. Robust Statistical Methods for Empirical Software Engineering. *Empirical Software Engineering* 22, 2 (April 2017), 579–630. doi:10.1007/s10664-016-9437-5

[38] Amy J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. 2006. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Transactions on Software Engineering* 32, 12 (2006), 971–987. doi:10.1109/TSE.2006.116

[39] Bonan Kou, Shengmai Chen, Zhijie Wang, Lei Ma, and Tianyi Zhang. 2024. Do large language models pay similar attention like human programmers when generating code? *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 2261–2284.

[40] Olga Kovaleva, Alexey Romanov, Anna Rogers, and Anna Rumshisky. 2019. Revealing the Dark Secrets of BERT. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan (Eds.). Association for Computational Linguistics, Hong Kong, China, 4365–4374. doi:10.18653/v1/D19-1445

[41] Stefano Lambiase, Gemma Catolino, Fabio Palomba, and Filomena Ferrucci. 2024. Motivations, Challenges, Best Practices, and Benefits for Bots and Conversational Agents in Software Engineering: A Multivocal Literature Review. *ACM Comput. Surv.* 57, 4, Article 93 (Dec. 2024), 37 pages. doi:10.1145/3704806

[42] Thanh Le-Cong, Bach Le, and Toby Murray. 2025. Memory-Efficient Large Language Models for Program Repair with Semantic-Guided Patch Generation. arXiv:2410.16655 [cs.SE] https://arxiv.org/abs/2410.16655

[43] Fengjie Li, Jiajun Jiang, Jiajun Sun, and Hongyu Zhang. 2025. Evaluating the Generalizability of LLMs in Automated Program Repair. arXiv:2503.09217 [cs.SE]

[44] Hongyan Li, Meng Yan, Weifeng Sun, Xiao Liu, and Yunsong Wu. 2023. A first look at bug report templates on GitHub. *Journal of Systems and Software* 202 (Aug. 2023), 111709. doi:10.1016/j.jss.2023.111709

[45] Hao Li, Haoxiang Zhang, and Ahmed E. Hassan. 2025. The Rise of AI Teammates in Software Engineering (SE) 3.0: How Autonomous Coding Agents Are Reshaping Software Engineering. arXiv:2507.15003 [cs.SE]

[46] Jiliang Li, Yifan Zhang, Zachary Karas, Collin McMillan, Kevin Leach, and Yu Huang. 2024. Do machines and humans focus on similar code? exploring explainability of large language models in code summarization. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*. 47–51.

[47] Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. 2024. Snapkv: Llm knows what you are looking for before generation. *Advances in Neural Information Processing Systems* 37 (2024), 22947–22970.

[48] Zhong Li, Chong Zhang, Minxue Pan, Tian Zhang, and Xuandong Li. 2024. AACEGEN: Attention Guided Adversarial Code Example Generation for Deep Code Models. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 1245–1257.

[49] Shanchao Liang, Nan Jiang, Yiran Hu, and Lin Tan. 2025. Can Language Models Replace Programmers for Coding? REPOCOD Says 'Not Yet'. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (Eds.). Association for Computational Linguistics, Vienna, Austria, 24698–24717. doi:10.18653/v1/2025.acl-long.1204

[50] Shanchao Liang, Nan Jiang, Shangshu Qian, and Lin Tan. 2025. WAFFLE: Fine-tuning Multi-Modal Model for Automated Front-End Development. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (Eds.). Association for Computational Linguistics, Vienna, Austria, 24786–24802. doi:10.18653/v1/2025.acl-long.1208

[51] Rensis Likert. 1932. A technique for the measurement of attitudes. *Archives of Psychology* 22, 140 (1932), 1–55.

[52] Kui Liu, Anil Koyuncu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. 2019. 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)*. 102–113. doi:10.1109/ICST.2019.00020

[53] Gianluigi Lopardo, Frederic Precioso, and Damien Garreau. 2024. Attention Meets Post-hoc Interpretability: A Mathematical Perspective. arXiv:2402.03485 [stat.ML] https://arxiv.org/abs/2402.03485

[54] Scott M. Lundberg and Su-In Lee. 2017. A unified approach to interpreting model predictions. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 4768–4777.

[55] Mary McHugh. 2012. Interrater reliability: The kappa statistic. *Biochemia medica : časopis Hrvatskoga društva medicinskih biokemičara / HDMB* 22 (10 2012), 276–82.

[56] Mary L. McHugh. 2013. The chi-square test of independence. *Biochemia Medica* 23, 2 (2013), 143–149. doi:10.11613/bm.2013.018

[57] Niklas Metzger, Christopher Hahn, Julian Siber, Frederik Schmitt, and Bernd Finkbeiner. 2022. Attention Flows for General Transformers. arXiv:2205.15389 [cs.LG] https://arxiv.org/abs/2205.15389

[58] Khalil Mrini, Franck Dernoncourt, Quan Hung Tran, Trung Bui, Walter Chang, and Ndapa Nakashole. 2020. Rethinking Self-Attention: Towards Interpretability in Neural Parsing. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, Online, 731–742. doi:10.18653/v1/2020.findings-emnlp.65

[59] Zheng Ning, Yuan Tian, Zheng Zhang, Tianyi Zhang, and Toby Jia-Jun Li. 2024. Insights into natural language database query errors: From attention misalignment to user handling strategies. *ACM Transactions on Interactive Intelligent Systems* 14, 4 (2024), 1–32.

[60] NLTK. 2025. https://www.nltk.org/

[61] Replication Package. 2026. https://doi.org/10.5281/zenodo.21381449

[62] Matteo Paltenghi, Rahul Pandita, Austin Z. Henley, and Albert Ziegler. 2024. Follow-Up Attention: An Empirical Study of Developer and Neural Model Code Exploration. *IEEE Transactions on Software Engineering* 50, 10 (2024), 2568–2582.

[63] Nikhil Parasaram, Huijie Yan, Boyu Yang, Zineb Flahy, Abriele Qudsi, Damian Ziaber, Earl T. Barr, and Sergey Mechtaev. 2025. The Fact Selection Problem in LLM-Based Program Repair. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering* (Ottawa, Ontario, Canada) *(ICSE '25)*. IEEE Press, 2574–2586. doi:10.1109/ICSE55347.2025.00162

[64] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. arXiv:2009.10297 [cs.SE]

[65] Sofia Serrano and Noah A. Smith. 2019. Is Attention Interpretable?. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, Anna Korhonen, David Traum, and Lluís Màrquez (Eds.). Association for Computational Linguistics, Florence, Italy, 2931–2951. doi:10.18653/v1/P19-1282

[66] Lin Shi, Fangwen Mu, Yumin Zhang, Ye Yang, Junjie Chen, Xiao Chen, Hanzhi Jiang, Ziyou Jiang, and Qing Wang. 2022. BugListener: identifying and synthesizing bug reports from collaborative live chats. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 299–311.

[67] Yu Shi, Abdul Ali Bangash, Emad Fallahzadeh, Bram Adams, and Ahmed E. Hassan. 2025. HAFix: History-Augmented Large Language Models for Bug Fixing. arXiv:2501.09135 [cs.SE] https://arxiv.org/abs/2501.09135

[68] Mozhan Soltani, Felienne Hermans, and Thomas Bäck. 2020. The significance of bug report elements. *Empirical Software Engineering* 25, 6 (Nov. 2020), 5255–5294.

[69] Sonar. 2025. https://www.sonarsource.com

[70] Stackoverflow. 2025. https://survey.stackoverflow.co/2025/technology

[71] Streamlit. 2025. https://streamlit.io/

[72] Caizhi Tang, Qing Cui, Longfei Li, and Jun Zhou. 2023. GINT: A Generative Interpretability method via perturbation in the latent space. *Expert Systems with Applications* 232 (Dec. 2023), 120570. doi:10.1016/j.eswa.2023.120570

[73] Graham J. G. Upton. 1992. Fisher's Exact Test. *Journal of the Royal Statistical Society. Series A (Statistics in Society)* 155, 3 (1992), 395–402. doi:10.2307/2982890

[74] Thomas Valentin, Ardi Madadi, Gaetano Sapia, and Marcel Böhme. 2025. Incoherence as Oracle-less Measure of Error in LLM-Based Code Generation. arXiv:2507.00057 [cs.PL] https://arxiv.org/abs/2507.00057

[75] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2023. Attention Is All You Need. arXiv:1706.03762 [cs.CL] https://arxiv.org/abs/1706.03762

[76] Chunqiu Steven Xia, Yifeng Ding, and Lingming Zhang. 2024. The Plastic Surgery Hypothesis in the Era of Large Language Models. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering* (Echternach, Luxembourg) *(ASE '23)*. IEEE Press, 522–534.

[77] Chunqiu Steven Xia, Zhe Wang, Yan Yang, Yuxiang Wei, and Lingming Zhang. 2025. Live-SWE-agent: Can Software Engineering Agents Self-Evolve on the Fly? arXiv:2511.13646 [cs.SE]

[78] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-Trained Language Models. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1482–1494.

[79] Chunqiu Steven Xia and Lingming Zhang. 2024. 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* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 819–831. doi:10.1145/3650212.3680323

[80] Boyang Yang, Luyao Ren, Xin Yin, Jiadong Ren, Haoye Tian, and Shunfu Jin. 2025. Input Reduction Enhanced LLM-based Program Repair. arXiv:2507.15251 [cs.SE]

[81] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. arXiv:2405.15793 [cs.SE]

[82] Yaopeng Yang, Chuanyi Li, Zhifeng Han, Rui Li, Kui Xu, Qingyuan Li, Wenkang Zhong, Zongwen Shen, Zhiwei Fei, Jidong Ge, and Bin Luo. 2025. Patch Generation in APR: A Survey from the Perspectives of Utilizing LLMs and Using APR-Specific Information. *ACM Trans. Softw. Eng. Methodol.* (Aug. 2025).

[83] Jialu Zhang, José Pablo Cambronero, Sumit Gulwani, Vu Le, Ruzica Piskac, Gustavo Soares, and Gust Verbruggen. 2024. PyDex: Repairing Bugs in Introductory Python Assignments using LLMs. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 133 (April 2024), 25 pages.

[84] Qingru Zhang, Chandan Singh, Liyuan Liu, Xiaodong Liu, Bin Yu, Jianfeng Gao, and Tuo Zhao. 2024. Tell Your Model Where to Attend: Post-hoc Attention

Steering for LLMs. arXiv:2311.02262 [cs.CL] https://arxiv.org/abs/2311.02262

[85] Yifan Zhang, Chen Huang, Zachary Karas, Thuy Dung Nguyen, Kevin Leach, and Yu Huang. 2025. *Enhancing Code LLM Training with Programmer Attention*. Association for Computing Machinery, New York, NY, USA, 616–620.

[86] Yifan Zhang, Chen Huang, Yueke Zhang, Jiahao Zhang, Toby Jia-Jun Li, Collin McMillan, Kevin Leach, and Yu Huang. 2025. EyeMulator: Improving Code Language Models by Mimicking Human Visual Attention. *arXiv preprint* (2025).

[87] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1592–1604.

[88] Zifan Zheng, Yezhaohui Wang, Yuxin Huang, Shichao Song, Mingchuan Yang, Bo Tang, Feiyu Xiong, and Zhiyu Li. 2024. Attention Heads of Large Language Models: A Survey. arXiv:2409.03752 [cs.CL] https://arxiv.org/abs/2409.03752

[89] Zhenhong Zhou, Haiyang Yu, Xinghua Zhang, Rongwu Xu, Fei Huang, Kun Wang, Yang Liu, Junfeng Fang, and Yongbin Li. 2025. On the Role of Attention Heads in Large Language Model Safety. arXiv:2410.13708 [cs.CL]

[90] Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg, Sascha Just, Adrian Schroter, and Cathrin Weiss. 2010. What Makes a Good Bug Report? *IEEE Trans. Softw. Eng.* 36, 5 (Sept. 2010), 618–643. doi:10.1109/TSE.2010.63