**Response Letter of Paper #1143**

Dear ICSE 2026 Program Committee and Reviewers,

Thank you for reviewing our submission, “XRFix: Exploring Performance Bug Repair of Extended Reality Applications with Large Language Models.” We received scores of 3 (Weak Accept), 2 (Weak Reject), and 2 (Weak Reject), and the metareview recommendation is **Major Revision**.

We appreciate your constructive feedback, which helped us improve the paper’s novelty and soundness. We have revised the manuscript accordingly, addressing all concerns. This letter outlines the comments and our responses, along with a highlighted version of the revised paper for easy reference.

Thank you for your support.

Sincerely,

Authors

**Response to Metareview**

|  |
| --- |
| This paper presents XRFix, a framework that leverages static analysis tools and large language models (LLMs) to repair performance bugs in Unity-based XR applications. The paper is clearly written, presents a timely and well-motivated problem in an emerging domain, and contributes a publicly available dataset for future research.  The authors’ rebuttal clarified several reviewer concerns, and the reviewers appreciate the willingness to incorporate improvements. However, several important methodological issues remain and must be addressed to meet ICSE’s standards:   * Baseline comparison: The current evaluation does not include any comparison with prior APR methods. Reviewers unanimously emphasized the importance of benchmarking XRFix against state-of-the-art baselines, not just legacy or Unity-specific tools. This comparison is critical to establish the broader relevance and performance of the approach. |
| Response:  Thanks for your suggestions.  To the best of our knowledge, there are no existing APR methods specifically developed for XR performance bugs. Therefore, for a fair comparison, we selected three state-of-the-art APR approaches that can handle general bugs and support C# (不支持XR特性的bug 3D scenes等): AlphaRepair, Fine-tuned CodeT5, and Self-Repair. These methods represent different strategies for applying large language models to APR: AlphaRepair is the first work to use CodeBERT to do infilling for APR task, Fine-tuned CodeT5 is trained for multi-hunk bug repair, and Self-Repair leverages LLMs’ feedback to guide the repair process. By comparing XRFix with these diverse approaches, we can more effectively demonstrate its performance.  We include our revision in Section 4.3 to introduce our APR baselines. Additionally, we have added a new research question (Section 4.10) to clarify our conclusions. |
| * Prompting strategy depth: The study currently relies on zero-shot prompting. To strengthen the contribution, reviewers recommend integrating few-shot, retrieval-augmented, or context-sensitive prompting strategies—or providing stronger justification for the zero-shot setup. |
| Thanks for your comments.  （选择别的方法存在困难，所以选择了one-shot）First, we establish a baseline by using zero-shot prompting to evaluate the performance of LLMs on our curated XR performance bug repair datasets. In our revised paper, we include **prompt e** to investigate one-shot prompting, where we provide LLMs with a bug and its corresponding fix as an example. Prompt e is introduced in Section 3.3. Additionally, we have added the related experiments and conclusions in Section 4.  We haven chosen one-shot learning because we want to adopt prompt engineering with token limits (max tokens = 4000). Our dataset includes three types of bug scenarios, which require us to provide enough context for the LLMs. This context, especially for class-level bugs, uses a large portion of the available tokens. Since we have already built bug repair datasets for XR performance bugs, we can explore other methods in future work such as RAG, agent-based methods and context-sensitive prompting. （各个方法的难点）One of our main contribution is the collection of real-world performance bugs and the creation bug repair datasets. Furthermore, To our knowledge, we are the first to evaluate LLMs’ ability to fix XR performance bugs using zero-shot and one-shot prompting. |
| * Prompt ablation depth: While some prompt variants are evaluated, the analysis remains shallow. A more detailed breakdown of how prompt components (e.g., bug localization, use of comments, repair hints) affect outcomes would enhance interpretability and generalizability. |
| Thanks for your comments.  We have performed an ablation study to analyze the impact of different prompt components in our templates. The revised results and conclusions are now presented in RQ4, Section 4.8. Our findings show that the **Bug Instruction** component provides the most significant improvement. In contrast, using an **Alternative Comment Style** can confuse LLMs and reduce their performance. （component详细介绍一下）  图像 |
| * Clarification of bug types: The boundary between bad practices and true performance bugs needs to be clarified, especially since the detection process reuses UnityLint. A clearer distinction would improve confidence in the benchmark’s scope and utility. |
| （原文中的bad practice指的是：…，引入论文这些practice会引入bug）We would like to clarify that **inefficient coding**（**IC**） **practices** used by developers can lead to various performance bugs. To prevent misunderstandings, we have revised the wording in the Abstract and Introduction section accordingly. We would like to note that UnityLint is primarily designed to detect bad smells in Unity projects, which differs significantly from the goals of our study. |
| * Statistical reporting: Statistical testing, such as the reported Wilcoxon results, should be integrated more clearly into the main experimental analysis. Including confidence intervals would also strengthen the claims. |
| Thank you for your feedback. We have added statistical testing to support our conclusions.  For CodeBLEU, since we need to aggregate the average scores of CodeBLEU and similarity scores, “*we report the average scores with confidence interval computed with bootstrap sampling with 1,000 resamples. ”* When comparing different CodeBLEU scores, we use the Wilcoxon signed-rank test to assess statistical significance. All related results in the Experiment section (Section 4) have been updated accordingly. We also introduce our settings in Section 4.2. |

**Response to Reviewer A**

|  |
| --- |
| **Paper summary** |
| The paper proposes XRFix, a framework that uses static analysis and LLMs to repair performance bugs in Unity-based XR applications. It builds a new dataset, designs multiple prompt templates, and evaluates repair success across several LLMs.  Strengths  + The paper addresses an important and underserved domain  + Construction of an open-source XR-specific performance bug dataset fills a significant empirical gap   * Clear paper organization with readable writing   Weaknesses  - Technical methodology is primarily adapted from existing APR and LLM repair approaches, with limited fundamental innovation.  - Inconsistency in motivation framing: the paper overstates the novelty gap (e.g., ignoring UnityLint in the abstract) and does not rigorously explain why prior APR approaches are unsuitable for XR. |
| Thank you for taking the time to review our paper and your valuable comments. We appreciate your feedback. Below, we address each of your comments in details. |
| **Detailed comments for authors** |
| 1. Novelty   * The paper addresses an underserved application domain XR (Extended Reality) performance bugs with LLM-driven repair, which is relatively novel. * The construction of an XR-specific bug dataset from open-source Unity projects is a meaningful contribution to the community. * While the application domain (XR) is new, the underlying approach (static analysis + LLM prompting + zero-shot repair evaluation) is very similar to several recent APR-with-LLMs papers (e.g., AlphaRepair, RepairAgent, LLM4BugFix) thus, the technical novelty is somewhat moderate, not radical. * The paper sometimes overstates the novelty by claiming "no accurate bug detection tools" despite acknowledging prior work like UnityLint. Similarly, prior APR approaches are dismissed without a clear technical rationale for why they cannot be adapted to XR contexts. |
| Thanks for your comments xxx  With regarding your concern of “xxx”, we have explanations: The XR domain lacks accurate detection tools for **performance bugs**. Unlike XRFix, UnityLint mainly targets bad coding practices and does not directly address performance issues in XR projects. To address this gap, we customized our SAT tools to better identify and detect performance bugs in XR applications.  Existing APR methods targeted in program codes fall short for XR applications, including diverse virtual objects and 3D scenes. Moreover, the scarcity of datasets limits APR's adaptation to XR. In our work, we first construct bug repair evaluation datasets for XR performance bugs. Then, our tool is proven to be efficient to handle bugs in both asset files (YAML) and script files (C#) with three bug scenarios.  What’s more, to make a fair comparison, we selected three state-of-the-art APR approaches that can handle general bugs and support C#: AlphaRepair, Fine-tuned CodeT5, and Self-Repair. These methods represent different strategies for applying large language models to APR: AlphaRepair is the first work to use CodeBERT to do infilling for APR task, Fine-tuned CodeT5 is trained for multi-hunk bug repair, and Self-Repair leverages execution feedback to guide the repair process. By comparing XRFix with these diverse approaches, we can conclude that it is hard for general APR approaches to adapt to XR settings.(原理上的解释为什么不能adapt)  We include our revision in Section 4.3 to introduce our APR baselines. Additionally, we have added a new research question (Section 4.10) to clarify our conclusions. |
| * The contribution claims in the abstract and introduction blur the line between domain novelty and methodological novelty, which could be clarified. |
| Thank you for the feedback. We've updated the claims in both the abstract and introduction. 摘要和introduction修改后加上怎么修改的 |
| 2. Rigor   * The paper demonstrates careful dataset curation with multiple stages of bug mining, classification, and validation. Multiple LLMs and prompt templates are evaluated systematically across bug granularities. * Bug detection correctness is assumed rather than formally validated: there is no measurement of precision, recall, or false positive rates for static analysis queries. Only anecdotal mention of “two authors checked” bugs without formal sampling or inter-annotator agreement. |
| * As discussed in RQ2, we use precision to evaluate bug detection tools. Two authors with professional XR development experience independently analyze all test results. Then they have a two-hour meeting to discuss. To verify how two authors agree on the detection result, we employ Cohen’s Kappa coefficient. The result is 0.83 (>0.8), indicating strong consistency. We add this score in our revision in Section 4.6. |
| * Misuse of terminology: the paper often conflates bad practices with true bugs , weakening the precision of claims. |
| We would like to clarify that **inefficient coding practices** used by developers can lead to various performance bugs. To prevent misunderstandings, we have revised the wording in the Abstract and Introduction section accordingly. |
| * Classification of bug complexity (Direct call, Cross-Function call, Cross-Class call) is hand-engineered without reported validation: no inter-rater reliability, no discussion of ambiguous cases. |
| * 考虑到XR的特性(Unity的high-level idea)来划分的类型, we classified bug complexity based on the location of data flows' source and sink provided by CodeQL(validation怎么验证,为什么没有ambiguous case). For example, in the case of cross-class call, the source and sink must be located in two different classes. We add this details in Section 3.1:   “*Since CodeQL can derive complex dataflows depending on the compilation process, we utilize it to validate these bugs from target repositories. There are three steps to perform CodeQL analysis for C# codes of Unity projects. First, we construct the CodeQL database by compiling C# projects. Second, we run CodeQL queries against the database. Last, we adopt CodeQL to interpret query results. Notably, we create CodeQL queries to analyze data flows and identify each bug across various code scenarios (details given in § 3.2). We determine each bug scenario based on the locations of the source and sink as identified by CodeQL. For example, in the case of cross-class call, the source and sink must be located in two different classes.*” |
| * Weak ablation studies: No evaluation of the impact of different prompt styles (beyond a brief mention of template comparisons). Could have tested if bug fix rates are causally linked to prompt differences. |
| We have performed an ablation study to analyze the impact of different prompt components in our templates. The revised results and conclusions are now presented in RQ4, Section 4.8. Our findings show that the **Bug Instruction** component provides the most significant improvement. In contrast, using an **Alternative Comment Style** can confuse LLMs and reduce their performance.  图像 |
| * Lack of statistical significance tests on repair success rates between LLMs or prompts. |
| Thank you for your feedback. We have added statistical testing to support our conclusions.  For CodeBLEU, since we need to aggregate the average scores of CodeBLEU and similarity scores, “*we report the average scores with confidence interval computed with bootstrap sampling with 1,000 resamples. ”* When comparing different CodeBLEU scores, we use the Wilcoxon signed-rank test to assess statistical significance. All related results in the Experiment section (Section 4) have been updated accordingly. We also introduce our settings in Section 4.2. |
| 3. Relevance   * Relevant SE problem   4. Verifiability and Transparency   * Prompt templates are only briefly summarized, and no full actual prompts or variable substitutions are provided. * Describe Tree-sitter parsing coverage, failure rates (if any), and how Unity-specific constructs were handled.   5. Presentation   * The overall structure of the paper is logical |
| * In Section 4.11, we provided a case study of GPT-4o instructed by prompt c to fix a class level bug. Also, we introduced the parameters settings for LLMs in Section 4.1. * For our dataset construction, we first used Tree-sitter to filter target open-source projects across various code scenarios. Next, we employed CodeQL to further locate potential bugs within these projects. Finally, each bug scenario was manually validated and curated to ensure the quality and accuracy of our dataset. We have clarified more details in Section 3.1 and we also include the syntax rules in our Anonymous Repo. |
| **Questions for authors’ response** |
| * Since only Unity XR projects were considered, can the authors clarify the extent to which their findings and methodology generalize to XR apps developed in other engines? |
| Our framework can be extended to other XR engines by 1) constructing bug datasets, 2) identifying bugs through tailored static analysis tools (SATs), and 3) utilizing LLMs to repair bugs with diverse prompts.（举例Unreal语言特性，没有工具分析3D scenes等文件） |
| * Were any statistical significance tests conducted to validate that the observed improvements between different prompt templates (e.g., Template c vs Template d) are meaningful rather than random? |
| As mentioned above, we have added statistical testing to support our conclusions.  For CodeBLEU, since we need to aggregate the average scores of CodeBLEU and similarity scores, “*we report the average scores with confidence interval computed with bootstrap sampling with 1,000 resamples. ”* When comparing different CodeBLEU scores, we use the Wilcoxon signed-rank test to assess statistical significance. All related results in the Experiment section (Section 4) have been updated accordingly. We also introduce our settings in Section 4.2.  （Template c vs Template d的结论） |

**Response to Reviewer B**

|  |
| --- |
| **Paper summary** |
| This paper introduces XRFix, an LLM-based automated bug-fixing framework for open-source Extended Reality (XR) applications. XR apps pose unique performance challenges due to complex operations like 3D rendering and real-time animation. XRFix tackles three key challenges: lack of XR bug datasets, bug detection tools, and effective repair tools. The authors build a dataset from 23 XR projects with 104 real-world bugs and adapt static analysis tools for bug detection. They design prompt strategies for fixing bugs at multiple code levels using LLMs. Experiments show GPT-4o achieves a 67.31% fix rate across 10 XR bug types.  Strengths  + Realistic XR performance‑bug dataset with 104 issues across 23 Unity projects  + Multi‑dimensional evaluation combining automated metrics, Unity‑editor tests, and manual review  + Fully documented, publicly released artifacts ensuring reproducibility  Weaknesses  - Statistical analysis, error bias  - Manual analysis details |
| Thank you for your valuable comments. We will address each of your questions in the following sections. |
| **Detailed comments for authors** |
| 1. Novelty   * Tailoring APR techniques to XR performance bugs and integrating static analyzers with LLM‐based patch generation fills a clear gap beyond generic APR research at the best of my knowledge. The domain‑specific dataset and detection rules represent a meaningful advance.   2. Rigor   * Authors report fix‐rate differences between models but never show any error bars or confidence intervals. They also don’t say the rationale behind the temperature, top‑k/top‑p, or token limits they used when calling each LLM, which means anyone trying to replicate their work won’t know the exact settings especially the rationale. Although they test prompts at different scopes (single line, function, class), they never break down which part of the prompt actually makes a difference — a simple ablation could have shown whether extra context helps or just confuses the model. The description of how they merge multi‑line patches is very brief, so it’s unclear how they handle overlapping edits or conflicts. Finally, their “manual check” step validates only a handful of patches, but they don’t explain how those patches were chosen or whether the inspectors agreed on the results.(意见拆分) |
| * Thanks for your advice, we have added statistical testing to support our conclusions. Regarding the fix rate, we follow Ref.[49] by querying each LLM five times, considering a bug fixed if any of the five samples pass the tests. We also compare the percentage of plausible fixes across all five attempts to minimize randomness.   For CodeBLEU, since we need to aggregate the average scores of CodeBLEU and similarity scores, “*we report the average scores with confidence interval computed with bootstrap sampling with 1,000 resamples. ”* When comparing different CodeBLEU scores, we use the Wilcoxon signed-rank test to assess statistical significance. All related results in the Experiment section (Section 4) have been updated accordingly. We also introduce our settings in Section 4.2.   * LLM Settings: The temperature setting aligns with conventional APR configurations, like ChatRepair. We set the max tokens to be 4000. (justification) For other parameters, we use the default values. We introduced our settings in Section 4.1. * Ablation Study for Prompt Templates: We have performed an ablation study to analyze the impact of different prompt components in our templates. The revised results and conclusions are now presented in RQ4, Section 4.8. Our findings show that the **Bug Instruction** component provides the most significant improvement. In contrast, using an **Alternative Comment Style** can confuse LLMs and reduce their performance. （换个说法叙述）   图像 |
| * Regarding to the concerns of multi-line Patch Merging: Thank you for your suggestion. （再介绍一下怎么做的）We have included more details in our revision in Section 4.1. In addition, we have updated Figure 2 in our framework to illustrate this process more clearly(介绍一下怎么改的).   In Section 4.1:  “*Different bug scenarios may cause the LLM to generate new code lines or functions. To merge these with the original code, we use Tree-sitter to extract function definitions and key variable declarations from each response and compare them to the original as shown in Figure 2. We compare the extracted AST features of buggy file and generated fix. If the response updates any bug-free functions, we replace the originals with the generated ones. We also replace buggy functions with their LLM-generated versions. However, if the LLM only provides code lines instead of full functions, we insert these lines right after the commented buggy function. After making these changes, we add the remaining code. This process produces a potentially LLM-fixed repository* ***plausible*** *fix. We carefully combine all content to ensure reliable evaluation results.*”   * Manual Evaluation: We selected SORL and AOBL because they can demonstrate the performance of bug repair both quantitatively and qualitatively. Also, the effects can be easily perceived visually. As outlined in Section 4.6.3, addressing the SORL bug can decrease FPS, whereas fixing the AOBL bug can enhance users' visual experience. We have added our explanation on the reason why we choose these two bugs in Section 4.7.3. |
| 3. relevance: The work aligns tightly with ICSE’s focus on automated debugging, software analysis, and emerging domains. XR application developers and SE researchers can find the methods and dataset directly applicable.  4. verifiability & transparency: The replication package is present and easy to understand.  5. presentation: Writing is clear and figures effectively illustrate the methodology, but the figures are too small and the text is very close together, making them hard to read. |
| Thank you for your valuable feedback regarding the figures and texts. We have revised the presentation by (increase font size提升图片的可读性)enlarging the figures and increasing the spacing between text elements to improve readability. We hope these changes make our contents clearer and easier to interpret. |

**Response to Reviewer C**

|  |
| --- |
| **Paper summary** |
| This paper presents XRFix, an LLM-based approach designed to automatically repair bugs in extended reality (XR) programs. The authors first construct a comprehensive codebase comprising 23 open-source XR projects and curate a dataset of 104 real-world XR-related bugs. Bug detection is performed using two static analysis tools, after which XRFix applies large language models to generate fixes at different granularities, including the single-file, function, and class levels. Experimental results demonstrate that the proposed approach shows promising effectiveness in repairing XR bugs.  Strengths  + The authors construct a dedicated open-source XR codebase and compile a dataset of real-world XR bugs.  + This paper proposes a comprehensive approach covering both bug detection and bug repair, specifically targeting bugs in augmented XR systems.  + The authors conducted extensive experiments to evaluate the effects of different LLMs, prompts, and levels of granularity.  Weaknesses  - Lack of comparison with baseline approaches  - Lack of in-depth analysis in RQ answers  - The novelty of the paper is somewhat limited, especially since it relies solely on zero-shot learning. |
| Thank you for your detailed review and constructive feedback. We address each of your comments below. |
| **Detailed comments for authors** |
| 1. Novelty   * One of the strengths of this paper is its contribution of a novel XR-specific bug dataset, along with proposed approaches for both bug detection and repair in XR systems. However, the repair method primarily relies on zero-shot prompting, which may limit its performance. A comparison with SOAT APR techniques would further strengthen the evaluation and highlight the effectiveness of the proposed approach. |
| Regarding xxx First, we establish a baseline by using zero-shot prompting to evaluate the performance of LLMs on our curated XR performance bug repair datasets. In our revised paper, we include **prompt e** to investigate one-shot prompting, where we provide LLMs with a bug and its corresponding fix as an example. Prompt e is introduced in Section 3.3. Additionally, we have added the related experiments and conclusions in Section 4.  We haven chosen one-shot learning because we want to adopt prompt engineering with token limits (max tokens = 4000). Our dataset includes three types of bug scenarios, which require us to provide enough context for the LLMs. This context, especially for class-level bugs, uses a large portion of the available tokens. Since we have already built bug repair datasets for XR performance bugs, we can explore other methods in future work such as RAG, agent-based methods and context-sensitive prompting. One of our main contribution is the collection of real-world performance bugs and the creation bug repair datasets. Furthermore, To our knowledge, we are the first to evaluate LLMs’ ability to fix XR performance bugs using zero-shot and one-shot prompting. |
| 2. Rigor:  The methodology and experimental design in this paper are generally rigorous. However, the following issues remain that could be further addressed to strengthen the rigor of the study:   * The paper lacks a comparison with baseline methods. Including evaluations against existing baseline approaches would help demonstrate the relative effectiveness and contribution of the proposed method. |
| Thanks for your suggestions.  To our knowledge, there are no existing APR methods specifically developed for XR performance bugs. Therefore, for a fair comparison, we selected three state-of-the-art APR approaches that can handle general bugs and support C#: AlphaRepair, Fine-tuned CodeT5, and Self-Repair. These methods represent different strategies for applying large language models to APR: AlphaRepair is the first work to use CodeBERT to do infilling for APR task, Fine-tuned CodeT5 is trained for multi-hunk bug repair, and Self-Repair leverages execution feedback to guide the repair process. By comparing XRFix with these diverse approaches, we can more effectively demonstrate its performance.  We include our revision in Section 4.3 to introduce our APR baselines. Additionally, we have added a new research question (Section 4.10) to clarify our conclusions. |
| * The related work section lacks a discussion of recent advances in APR, particularly agent-based approaches. Given the growing relevance of LLM-powered agent systems in software engineering tasks, including bug detection and repair, it would strengthen the paper to include a comparative analysis or at least a discussion of such methods. This would help contextualize the novelty of the proposed approach and clarify how it differs from or builds upon these recent trends. |
| Thank you for your suggestion. We have revised the Related Work section to include more recent advances in APR, particularly focusing on how recent studies have incorporated LLMs into APR tasks. Specifically, we now discuss agent-based APR methods such as FixAgent, RepairAgent, and Agentless, which enable LLMs to assume multiple roles and interact with external tools to facilitate the repair process. In our study, we explore the capabilities of LLMs with zero-shot and one-shot prompting for XR performance bugs. For future work, we plan to further enhance LLM performance by incorporating agent-based approaches. For example, we are considering replacing our current SAT tools with LLMs for bug localization.(agent-based难点，LLM针对XR场景没有bug detection的有效方法) |
| * In Section 4.6, the paper states that “Overall, our customized CodeQL queries can detect 57.7% of our collected bugs with the precision of 100%.” Could the authors clarify how this 57.7% detection rate was calculated? It is currently unclear how the mapping between the collected bugs and the CodeQL-detected bugs was established, and what criteria were used to determine a successful detection. |
| To clarify more clearly, we have revised Table 5 to present the combined results of our SAT tools. As described in Section 3.2, we use UnityLint and CodeQL to locate bugs for constructing our bug repair evaluation dataset. To assess the performance of these SAT tools, we employ precision metrics based on manual evaluation. Our results indicate that, using existing rules, we identified ten false positives, whereas our customized rules yielded no false positives. Overall, the SAT tools achieved a high precision of 90.4%. We have updated Section 4.6 to clearly reflect these revisions.  贴上表 |
| 3. Relevance  This paper addresses the important problem of bug repair in XR systems, which is a critical and emerging challenge in modern software engineering. The topic is highly relevant to the field, particularly in the context of maintaining and improving the reliability of XR applications.  4. Verifiability and Transparency  The authors have demonstrated a strong commitment to verifiability and transparency by publicly releasing their source code, dataset, and detailed usage instructions. This allows other researchers to thoroughly examine, reproduce, and potentially build upon the proposed method. |
| 5. Presentation  The overall structure of the paper is clear and logically organized. The writing is generally easy to understand and follow, which helps readers grasp the key contributions and methodology without significant difficulty.  RQ answer lacks sufficient depth in discussion and interpretation. For instance, in RQ1, while the authors provide statistics for different categories of bugs, there is little explanation or analysis regarding the underlying causes or implications of the observed distributions. A more thorough discussion would enhance the readers' understanding of the findings and strengthen the contribution of the study. |
| Thanks for your suggestions. Regarding your concern of analysis of RQ:  We have revised RQ1 to conduct in-depth analysis about the distribution of different bugs and three different bug scenarios.  In Section 4.5: “*We observe from Figure 5(a) that the IDU bug occupies the largest amount among all bugs, while SORL, AOBL, and RS have the smallest ones. Also, we can conclude that IDU and RWT are the most universal bugs among the XR projects. This suggests that these two bugs are pervasive challenges for XR developers.*  *Figure 5(b) shows the distributions of bugs in three types of bug scenarios, demonstrating the prevalence of different bug scenarios. Among them, single-line level bug scenarios have the highest frequency of 76, i.e., more than 73.07%. In contrast, function-level and class-level bug scenarios have frequencies of 22 and 6, respectively. The characteristics of XR's real-time rendering and frequent interactions make single-line performance bugs visible. Bugs IDU and RWT consist of all three code scenarios. For bug NAU, it only contains two code scenarios: single-line level and class-level. The other bugs primarily feature only one scenario. As a result, it is essential for XRFix to handle diverse code scenarios to address performance bugs in XR projects.*” |
| **Questions for authors’ response** |
| 1. In Section 4.6, when comparing the performance of different LLMs, the results vary across different evaluation metrics. Could the authors provide further discussion on which LLM would be more suitable in real-world applications? For instance, how should practitioners choose an appropriate LLM when the evaluation metrics lead to different conclusions? The same to the choice of prompts. |
| Thanks for your advice. We have highlighted key considerations for practical applications in Section 7. Our revisions are as follows:  “*To inform practical applications, we highlight several key considerations: (1) Efficiency: General LLMs generally demonstrate superior performance compared to Code LLMs, with GPT-4o surpassing GPT-3.5-Turbo in effectiveness. (2) Bug Type Specificity: GPT-3.5-Turbo should be avoided for addressing complex, class-level bugs due to its limitations in handling such cases. (3) Code LLM Performance: Among Code LLMs, Deepseek-Coder consistently yields more dependable results than both Code Llama and StarChat-beta. (4) Prompt Design: Prompt c tends to generate more plausible code fixes, whereas prompt d produces more reliable outputs. These factors should be carefully considered by practitioners when selecting models and prompt strategies.*” |
| 2. The study focuses on zero-shot prompting for applying LLMs. Have the authors considered exploring more advanced prompt engineering strategies, such as few-shot prompting or retrieval-augmented generation (RAG)? These approaches may potentially lead to better bug-fixing performance. It would be helpful to understand why these strategies were not included. |
| Thanks for your suggestions.  As mentioned above, to our knowledge, there are no existing APR methods specifically developed for XR performance bugs. Therefore, for a fair comparison, we selected three state-of-the-art APR approaches that can handle general bugs and support C#: AlphaRepair, Fine-tuned CodeT5, and Self-Repair. These methods represent different strategies for applying large language models to APR: AlphaRepair is the first work to use CodeBERT to do infilling for APR task, Fine-tuned CodeT5 is trained for multi-hunk bug repair, and Self-Repair leverages execution feedback to guide the repair process. By comparing XRFix with these diverse approaches, we can more effectively demonstrate its performance.  We include our revision in Section 4.3 to introduce our APR baselines. Additionally, we have added a new research question (Section 4.10) to clarify our conclusions. |
| 3. In Section 4.5, the paper states that “Overall, our customized CodeQL queries can detect 57.7% of our collected bugs with the precision of 100%.” Could the authors clarify how this 57.7% detection rate was calculated? |
| To clarify more clearly, we have revised Table 5 to present the combined results of our SAT tools. As described in Section 3.2, we use UnityLint and CodeQL to locate bugs for constructing our bug repair evaluation dataset. To assess the performance of these SAT tools, we employ precision metrics based on manual evaluation. Our results indicate that, using existing rules, we identified ten false positives, whereas our customized rules yielded no false positives. Overall, the SAT tools achieved a high precision of 90.4%. We have updated Section 4.6 to clearly reflect these revisions. |
| 4. What potential impact could this research have on future work in XR bug detection and repair? Are there specific directions or applications that the authors envision this work enabling or inspiring? |
| We can summarize our impacts into three aspects:  First, researchers can utilize our constructed bug repair datasets and uses our tool as baselines to further enhance the performance of bug repair in XR domain.  Second, they can extend our XRFix to other bugs of XR apps developed by other engines. Following our ways, they can employ static analysis tools to construct a comprehensive bug repair dataset.  Furthermore, our prompt templates can be enhanced to effectively guide LLMs in bug‐fixing.  As for future work, we'll extend our framework to other XR engines and incorporate advanced techniques such as RAG and agents. |