**Response Letter of Paper ######**

Dear ICSE’25 PC and Reviewers:

Regarding our previous submission entitled “Defining and Detecting Virtual and Augmented Reality Performance Bugs with LLMs” to your ICSE 2025, we received the following scores: **3 (Weak accept), 2 (Weak reject), and 3 (Weak accept)** from three reviewers. The metareview recommendation is **Major Revision**. Many thanks for your comments and positive evaluations of our paper. Let us express our sincere thanks to all the reviewers and PC. Your constructive comments guide us to further improve this paper in terms of novelty and soundness.

We have carefully revised the paper by addressing all your concerns. In this response letter, we first list your concerns in a point-by-point manner. We then present our responses accordingly. Moreover, we also present a highlighted version of the revised paper for your reference so that you can quickly identify our revised parts.

Your sincerely,

Authors

**Point-by-Point Response to Metareview**

|  |
| --- |
| Thanks for the submission and the rebuttal. The reviewers agree that this paper addresses an important problem and the proposed dataset could be useful.  However, they still have some concerns after reading the authors’ response. They would like the authors to address the following points, before they make a decision about the paper:  \* a clear comparison with an existing general DL-based bug detection approach (fine-tuned or trained on the VR performance issue data). |
| **Response**: Thanks for your comment.  For a fair comparison, as for LLM models we fine-tune them based on our training set with the same hyperparameter setting and then evaluate the performance. Referring to the non-LLM model, we also train the model with the same hyperparameter setting based on our dataset.  In the revised paper, we have supplemented the additional details.  We excerpted the major revisions as follows.    In Section IV-C:  “For a fair comparison, we fine-tune other LLM models based on our training set with *the same hyperparameter settings* and then evaluate their performance. Regarding the non-LLM model (i.e., Word2vec), we also train the model based on our dataset with the *same hyperparameter settings*.” |
| \* the methodology section should also be updated to include more details. |
| **Response**: Thanks for your comment.  In the revised paper, we supplemented more details about data labeling in the methodology according to the concerns of reviewers.  We excerpted the major revisions as follows.  In Section III-C:  “We first obtain the small set of ground-truth labels based on the previous study [5]. These performance optimization commits are categorized into the first seven types as defined in § III-B by extracting the before-commit data. Then, the manually labeled seven categories of performance bugs (except for ``COMPA'' bugs) are obtained.”  In Section III-C:  “Regarding “COMPA” bugs, we search in the commit messages of the collected dataset by using the keywords “compatibility” and “compat” (i.e., wildcard expression) and tag them as compatibility-related data. To ensure the correctness of labeling, we invite two well-experienced student volunteers with 1 year of Unity and C# development experience to verify the correctness of the labeling. Only if both volunteers agree that the labeling is correct, we confirm the data as compatibility bug data.” |
| \* discuss the differences between the proposed method for detecting performance bugs in VR/AR applications and traditional bug detection methods. |
| **Response**: Thanks for your comment.  The proposed method (namely *PerfDetector*) distinguishes itself from traditional deep learning (DL)-based bug detectors in several perspectives. 1) *PerfDetector* considers not only generic code characteristics (such as source code and AST) but also object invocations in its inputs so that it can address the unique challenges of VR/AR projects. Although object invocation or method invocation has been adopted for bug detection in previous research, our method originally includes *unique features* of VR/AR scenes. These features include virtual object-related invocations and usage, which do not exist in traditional apps (e.g., mobile apps). As a result, these three types of inputs construct the original semantic and syntactic information of VR/AR projects, meanwhile it emphasizes the unique features of VR/AR scenes. 2) Our *PerfDetector* specifically considers the programming language characteristics of VR/AR projects. Specifically, we adopt an LLM pre-trained on C# code (Code-T5), where C# has been the predominant language in Unity VR/AR projects. This design greatly enhances the detector's performance over other LLM-based methods. For a fair comparison, we have trained and evaluated off-the-shelf methods with the same hyperparameter setting based on our dataset in RQ2.  In the revised paper, we elucidate the novelty and contributions of the proposed method by revising the corresponding parts in the contribution description, methodology, and experimental results. Please refer to the highlighted version of the revised paper for more details. We excerpt the major revisions as follows.  In Section I:  “  • We collect 1,013 open-source popular VR/AR repositories and construct a performance-bug dataset containing 10,950 function-level buggy samples for performance-bug detection.  …  • With the usage of labeled data, we propose an automatic performance bug detector based on CodeT5 to encode buggy AST, object invocations (especially APIs related to virtual object invocation), and buggy source code. The combination of these features is relevant to VR/AR app performance. *To the best of our knowledge, it is the first LLM-based tool to detect VR/AR performance bugs.* ”  In Section III-D:  “Although object invocation or method invocation has been adopted for bug detection in previous research [27], [28], our method originally includes unique features of VR/AR scenes. These features include virtual object-related invocations and usage, which do not exist in traditional apps (e.g., mobile apps). For example, Rigidbody-related APIs adopt a physics engine to control virtual objects. Collision-related and Colliding-related APIs can be used to control virtual objects to react to overlapping with or without physics effects, respectively [10]. RenderTexture-related APIs are used to create a life-like vision on a screen so as to enable immersive experiences.  ...  We extract these object invocations to emphasize the semantics of these APIs, as they are not included in the analysis of traditional mobile apps (since they do not emphasize the need for immersive experiences). This consideration sharply differentiates our proposed methods from traditional methods.  ...  By collecting these three types of inputs, we extract the original semantic and syntactic information of VR/AR project codes with emphasis on the unique features of VR/AR scenes, thereby helping LLMs to obtain comprehensive characteristics of VR project codes.  ”  [27] L. Sampaio and A. Garcia, “Exploring context-sensitive data flow analysis for early vulnerability detection,” Journal of Systems and Software, vol. 113, pp. 337–361, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121215002873  [28] S. Wang, T. Liu, and L. Tan, “Automatically learning semantic features for defect prediction,” in Proceedings of the 38th International Conference on Software Engineering, ser. ICSE ’16. New York, NY, USA: Association for Computing Machinery, 2016, p. 297–308. [Online]. Available: https://doi.org/10.1145/2884781.2884804  In Section IV-C:  “For a fair comparison, we fine-tune other LLM models based on our training set with the *same hyperparameter settings* and then evaluate their performance. Regarding the non-LLM (i.e., Word2vec), we also train the model based on our dataset with the *same hyperparameter settings*.” |

**Point-by-Point Response to Reviewer 1**

|  |
| --- |
| **Paper summary** |
| This paper studies performance bugs in virtual and augmented reality (VR/AR) applications, an emerging area in software engineering. The authors collects 10,950 buggy functions from 1,013 open-source VR/AR projects to define eight categories of performance bugs in VR/AR applications. Then, the authors propose a bug-detection method based on CodeT5 LLM, namely PerfDectector, which can learn from abstract syntax tree (AST), object invocation, and source code to detect performance bugs. Experimental results demonstrate that PerfDectector outperforms existing methods.  Strengths  + The paper provides good descriptions of performance bugs in VR/AR applications.  + Efforts to collect performance bugs in VR/AR applications are commendable.  + The source code and data are available for further research.  Weaknesses  - The manual labeling of performance bugs may introduce bias.  - The ablation study could be more comprehensive. |
| **Response**: Many thanks for your time reading this paper. We appreciate your positive and constructive comments on this paper.  We present the detailed response to your comments as follows. |
| Detailed comments for authors  Overall, I found this paper interesting. The authors conducted a comprehensive study on performance bugs in VR/AR applications and proposed a bug-detection method based on CodeT5 LLM. The main contributions of this paper are the definition of performance bugs in VR/AR applications. The paper is generally well-written. However, the manual labeling of performance bugs is not well justified, and the ablation study could not demonstrate the contribution of each component in PerfDectector.  My detailed comments are as follows.  1. The authors discussed the motivation and background of the study well. The authors claimed that traditional bug detection methods are difficult in detecting increasingly complex performance bugs in VR/AR apps. I would like to see more evidence or examples/citations to support this claim. |
| **Response**: Thanks for your comment. Unlike traditional bugs, VR apps are typically developed using frameworks like Unity to enable immersive experiences with the adoption of the Unity SDK. Consequently, the development of VR apps may invoke complex functions and diverse APIs, e.g., intricate object rendering and processing (as shown in Figure 8) and RigidBody API. Additionally, VR/AR scenes also involved multiple invocations of virtual objects, such as those shown in Figure 11. Unfortunately, traditional bug detection methods do not focus on these code features (even ignore them). In the revised paper, we add the related description in Section I “Introduction”. The revised texts are shown as follows.  In Section I, Paragraph 4:  “Although traditional bug detection methods can detect some bugs by static analysis [7], [8] and dynamic testing [9], they become struggling in detecting increasingly complex performance bugs in VR/AR apps. There are two reasons that traditional methods have difficulty in detecting performance bugs. Firstly, VR apps have been developed on top of game development engines, e.g., Unity’s game engine with the adoption of the Unity SDK. As a result, VR apps have typically used complex and diverse APIs unlike other apps only relying on simple SDKs. For example, Rigidbody APIs [10] have been typically adopted to control virtual objects in VR apps. In the later part of this paper (Fig. 8 in § III-B), we will further illustrate other code examples, which may involve complex image rendering/processing operations on VR objects. Secondly, in the VR/AR scenes, multiple virtual objects can be invoked by one method (details to be given in Fig. 11 in § III-D). Unfortunately, traditional bug detection methods cannot effectively detect these new yet complex bugs in VR/AR apps.”  [10] D. E. Rzig, N. Iqbal, I. Attisano, X. Qin, and F. Hassan, “Virtual Reality (VR) Automated Testing in the Wild: A Case Study on Unity-Based VR Applications,” in Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ser. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 1269–1281. [Online]. Available: <https://doi.org/10.1145/3597926.3598134> |
| 2. This paper requires some manual labeling efforts to categorize performance bugs in VR/AR applications. However, the authors did not provide enough details on how they label the performance bugs. How many annotators are involved in the labeling process? How do they ensure the quality of the labeling? And, what is the inter-annotator agreement? The authors should provide more details on this to minimize the bias introduced by manual labeling. |
| **Response**: Thanks for your comment. We next elucidate the labeling process as follows.  Regarding the first seven types of vulnerabilities as shown in Table I, we define performance vulnerability labels based on previous paper [5] and tag them to the collected data. In this paper, we newly define the eighth type of performance bug data (COMPA) by searching through compatibility-related keywords in the commit messages. Meanwhile, to ensure the correctness of labeling, we also invited two well-experienced student volunteers with 1 year of Unity and C# development experience to verify the correctness of the labeled data. Only if both volunteers agree that the data was correct, we labeled the data as COMPA. Consequently, we obtained 635 COMPA bug data.  In the revised paper, we supplemented additional details on the labeling process. Please refer to Section III-C and Section IV-A for more details. We excerpted the main revised text as follows:  In Section III-C:  “Regarding “COMPA” bugs, we search in the commit messages of the collected dataset by using the keywords “compatibility” and “compat” (i.e., wildcard expression) and tag them as compatibility-related data. To ensure the correctness of labeling, we invite two well-experienced student volunteers with 1 year of Unity and C# development experience to verify the correctness of the labeling. Only if both volunteers agree that the labeling is correct, we confirm the data as compatibility bug data.”  In Section IV-A:  “Before applying the proposed data labeling based on semi-supervised learning, we define eight categories of performance bugs as specified in § III-B and obtain 1,421 initially labeled samples (including 635 manually labeled COMPA bug samples).” |
| 3. The ablation study could be more comprehensive. I suggest that the authors remove one component at a time to demonstrate the contribution of each component in PerfDectector. For example, comparisons between the full model (PerfDectector) and variants such as PerfDectector-without-AST, PerfDectector-without-object-invocation, and PerfDectector-without-source-code are necessary to demonstrate the contribution of each component. |
| **Response**: Thanks for your comment.  In the revised paper, we improved ablation evaluation by implementing two additional models: *PerfDectector*-w/o-AST and *PerfDectector*-w/o-object-invocation models, where ‘w/o’ refers to ‘without’. In this way, we can better evaluate the contributions by different modules. We replotted Figure 13 (i.e., Figure 14 in the previous version) by updating the experimental results and adding a related description in RQ4. We excerpted the main revision as follows:  In Section IV-E:  “We use “AST only”, “*PerfDectector without (w/o) Source Code (SC)*”, “*PerfDectector without AST*”, “*PerfDectector without Object Invocation (OI)*” as inputs to train the model and compare their performance with our baseline model. Fig. 13 plots the results. It can be found that our *PerfDectector* outperforms the methods with the removal of different components, thereby further confirming the importance of three input sequences. In particular, the comparison between *PerfDectector* and “*PerfDectector without Source Code*” indicates that the introduction of source code information also has 14.73%, 12.81%, 14.21%, and 13.52% enhancements for accuracy, precision, recall, and F1, respectively. Moreover, the comparison between *PerfDectector* and “*PerfDectector without AST”* indicates that the introduction of AST information has 4.39%, 0.12%, 7.41%, and 3.78% enhancements for accuracy, precision, recall, and F1, respectively. Further, the comparison between *PerfDectector* and “*PerfDectector without Object Invocation*” indicates that the introduction of object call information has 5.08%, 0.57%, 7.82%, and 4.21% enhancements for accuracy, precision, recall, and F1, respectively. *PerfDectector* also outperforms 27.70%, 32.86%, 23.61%, and 28.20% in accuracy, precision, recall, and F1 than “AST only” input.” |
| Questions for authors’ response  1. How many annotators are involved in the labeling process? |
| **Response**: Thanks for your question.  We invited two student volunteers, each with 1+ year of Unity and C# development experience to participate in labeling. |
| 2. How do the annotators ensure the quality of the labeling? How to handle the situation where there is a disagreement among annotators? |
| **Response**: Thanks for your question.  For the first seven types of vulnerabilities in Table I, we defined performance vulnerability labels and tag them based on performance optimization ground-truth labels as defined by [5]. As for labeling COMPA bug (the last type), we labeled the data as COMPA bug only if both volunteers agreed that the data was correct. Consequently, we obtained 635 COMPA bug data. |

**Point-by-Point Response to Reviewer 2**

|  |
| --- |
| **Paper summary** |
| This paper proposes a novel approach to detect performance bugs in VR/AR software. In particular, the project labels an existing dataset use manual and semi-supervised approach, and then leverages a transformer-based model to predict whether a function contains performance bugs. The evaluation shows that the approach achieves about 50% f-score and outperforms existing general purpose bug detection approaches on the dataset.  Strengths  + Research on the performance analysis and issues of VR/AR software is very important and timely  + The paper did some good work on labelling and enhancing existing data  + The proposed approach is compared with multiple existig approaches  Weaknesses  - Limited novelty  - Missing evaluation of important parts  - Flawed evaluation setup  - Unclear evaluation process |
| **Response**: Many thanks for your positive evaluation and constructive comments on this paper. To address your concerns, we have provided the following point-by-point responses. |
| Detailed comments for authors  -----------------------------  I think the paper is a good initial step towards solving an important problem, but there are several issues need to be solved before the work can be published in top venues.  - Technique-wise, I do not see much novelty and adaptation targeting VR/AR software. The major part that is specific to VR/AR seems to be the dataset and corresponding labels. It is expected that training a deep-learning-based approach on a specific type of data will achieve better results on the same type of data. Therefore, the work can be viewed as an application and evaluation of existing code learning approaches to a specific dataset, so the novelty is limited. The evaluation results show that the approach is just slightly better than off-the-shelf approaches, so I think there are still large rooms to enhance. Furthermore, it is also not clear whether training the off-the-shelf approaches on the same dataset will lead to similar results as the proposed approach (I do not see much technical difference between them). |
| **Response**: Thanks for your comment. Let us further explain the key innovations of this paper and clarify your concerns as follows.  The *key innovations* of this paper are *threefold*. *First*, we investigate performance bugs in VR/AR, which is an emerging area in software engineering (SE). The automatic detection of VR/AR performance bugs is a quite new research area in SE while this research problem has not been extensively studied due to the lack of datasets and accurately extracted code features. *Second*, we construct a large-scale VR/AR performance bug dataset by identifying eight different types of bugs and using semi-supervised learning for efficient labeling. This well-labeled dataset is crucial for training bug-detection methods (not only for our proposed method but also other researchers in the SE community). *Third*, we present a novel bug-detection method for VR/AR performance issues based on the state-of-the-art large language models (LLMs).  The proposed method (namely *PerfDetector*) distinguishes itself from traditional deep learning (DL)-based bug detectors in several perspectives. 1) *PerfDetector* considers not only generic code characteristics (such as source code and AST) but also object invocations in its inputs so that it can address the unique challenges of VR/AR projects. Although object invocation or method invocation has been adopted for bug detection in previous research, our method originally includes *unique features* of VR/AR scenes. These features include virtual object-related invocations and usage, which do not exist in traditional apps (e.g., mobile apps). As a result, these three types of inputs construct the original semantic and syntactic information of VR/AR projects, meanwhile it emphasizes the unique features of VR/AR scenes. 2) Our *PerfDetector* specifically considers the programming language characteristics of VR/AR projects. Specifically, we adopt an LLM pre-trained on C# code (Code-T5), where C# has been the predominant language in Unity VR/AR projects. This design greatly enhances the detector's performance over other LLM-based methods. For a fair comparison, we have trained and evaluated off-the-shelf methods with the same hyperparameter setting based on our dataset in RQ2.  In the revised paper, we elucidate the novelty and contributions of the proposed method by revising the corresponding parts in the contribution description, methodology, and experimental results. Please refer to the highlighted version of the revised paper for more details. We excerpt the major revisions as follows.  In Section I:  “  • We collect 1,013 open-source popular VR/AR repositories and construct a performance-bug dataset containing 10,950 function-level buggy samples for performance-bug detection.  …  • With the usage of labeled data, we propose an automatic performance bug detector based on CodeT5 to encode buggy AST, object invocations (especially APIs related to virtual object invocation), and buggy source code. The combination of these features is relevant to VR/AR app performance. *To the best of our knowledge, it is the first LLM-based tool to detect VR/AR performance bugs.”*  In Section III-D:  “Although object invocation or method invocation has been adopted for bug detection in previous research [27], [28], our method originally includes unique features of VR/AR scenes. These features include virtual object-related invocations and usage, which do not exist in traditional apps (e.g., mobile apps). For example, Rigidbody-related APIs adopt a physics engine to control virtual objects. Collision-related and Colliding-related APIs can be used to control virtual objects to react to overlapping with or without physics effects, respectively [10]. RenderTexture-related APIs are used to create a life-like vision on a screen so as to enable immersive experiences.  ...  We extract these object invocations to emphasize the semantics of these APIs, as they are not included in the analysis of traditional mobile apps (since they do not emphasize the need for immersive experiences). This consideration sharply differentiates our proposed methods from traditional methods.  ...  By collecting these three types of inputs, we extract the original semantic and syntactic information of VR/AR project codes with emphasis on the unique features of VR/AR scenes, thereby helping LLMs to obtain comprehensive characteristics of VR project codes.  ”  [27] L. Sampaio and A. Garcia, “Exploring context-sensitive data flow analysis for early vulnerability detection,” Journal of Systems and Software, vol. 113, pp. 337–361, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121215002873  [28] S. Wang, T. Liu, and L. Tan, “Automatically learning semantic features for defect prediction,” in Proceedings of the 38th International Conference on Software Engineering, ser. ICSE ’16. New York, NY, USA: Association for Computing Machinery, 2016, p. 297–308. [Online]. Available: https://doi.org/10.1145/2884781.2884804  In Section IV-C:  “For a fair comparison, we fine-tune other LLM models based on our training set with the *same hyperparameter settings* and then evaluate their performance. Regarding the non-LLM (i.e., Word2vec), we also train the model based on our dataset with the *same hyperparameter settings*.” |
| - That being said, the dataset itself can be a good contribution. However, the semi-supervised learning is never evaluated. RQ1 asks on the effectiveness of the proposed semi-supervised labeling approach, but the evaluation just shows the distribution. That is not an evaluation on the effectiveness. It is also not clear why about 1400 samples were manually labelled. How do you decide this number and how are data points to be manually labeled selected? |
| **Response**: Thanks for your comment.  Manually labeling a large number of VR/AR performance bugs is extremely labor-intensive. To tackle this challenge, we proposed a semi-supervised method that leverages a small set of ground-truth labels to annotate the dataset first. Then, a data-labeling algorithm (Algorithm 2) was proposed to automatically label data samples.  Regarding your concern on the effectiveness of the proposed method, we have quantitatively evaluated the proposed semi-supervised learning approach from the following perspectives. 1) we randomly selected 825 samples (with identical probability) to verify clustering accuracy since the semi-supervised approach’s reliability depends on clustering accuracy. The selected samples have achieved 97.25% accuracy, thereby validating the reliability of our semi-supervised method. To clarify this issue, we provided more details in RQ1 and also reported it in Threats to Internal Validity (Section V). 2) it is worth mentioning that the first seven types of vulnerabilities in Table I were defined by previous paper [5]. The eighth type of bugs (COMPA) was defined by us. Specifically, we collected the commits by searching through compatibility-related keywords in the commit messages. To verify the correctness of the labeled data, we invited two well-experienced student volunteers with at least 1 year of Unity and C# development experience. Only if both volunteers agreed that the data was correct, we labeled the data as compatibility bug data. Thus, we get 635 COMPA bug data.  In the revised paper, we supplemented additional details in Section III-C and Section IV-A. We excerpted the mainly revised text as follows:  In Section III-C:  “Regarding “COMPA” bugs, we search in the commit messages of the collected dataset by using the keywords “compatibility” and “compat” (i.e., wildcard expression) and tag them as compatibility-related data. To ensure the correctness of labeling, we invite two well-experienced student volunteers with 1 year of Unity and C# development experience to verify the correctness of the labeling. Only if both volunteers agree that the labeling is correct, we confirm the data as compatibility bug data.”  In Section IV-A:  “Before applying the proposed data labeling based on semi-supervised learning, we define eight categories of performance bugs as specified in § III-B and obtain 1,421 initially labeled samples (including 635 manually labeled COMPA bug samples).” |
| The evaluation setup has clear threats to construction validity. 1500 functions randomly selected from the projects were used as negative samples. This is smaller than the positive samples used. Therefore, this evaluation dataset has a different distribution of data from the real-world, as in real world, there are typically many more normal functions than buggy functions. I think the work should consider the whole codebase except for the identified buggy functions or something in similar scale as negative samples. |
| **Response**: Thanks for your comment.  The reason that we collected 1500 bug-free data as negative data lies in the **multi-class classification** (not binary classification) model in our detector. Our goal is to construct a model to detect multiple types of performance bugs rather than construct multiple binary classification models to detect multiple bugs. To this end, we aimed to balance the amount of negative data and the amount of data in each positive class as much as possible. We focus on the model's performance in each class of positive and negative bugs as equally as possible. So, equal attention needs to be paid to each category during the model training. If the amount of negative data is chosen to be the same as that of positives, this setting essentially converts the problem into a binary classification, in which the model will focus on detecting negative data, thereby deviating from our goal. To further clarify this issue, we supplemented new experiments by adding the amount of negative data to be consistent with the amount of positive data of all types. We find that the result is not better than our approach. The details are shown in Section IV-D.  Section IV-D:  “In particular, since we focus on constructing a multi-class classification model in our detector, we aim to balance the amount of negative data and the amount of data in each positive class as much as possible. In order to validate the rationality of the dataset distribution settings, we supplement a new experiment by adding the amount of negative data to be consistent with the amount of positive data of all types. We name it *Balanced Negatives and Positives* (Balanced NP) data. The result is shown in Table III. We find that the total result is not better than *PerfDetector* (2.55% less than *PerfDetector* in F1). *PerfDetector* performs better in the detection of most types of performance bugs. This also means that our model achieves better performance with less training time (because of the smaller amount of data).” |
| - Part of the evaluation process is unclear. For example, what does the precision and recall mean in this paper? Do they refer to a single category of VR/AR issues? If so, why not reporting the precision and recall for each category? Or do they refer to just a boolean value whether a code piece is a performance bug? If so, how do you use the category labels in the evaluation? Also, I did not find how do you separate training and testing data. Did you do cross-validation or chronological separation? What was the ratio? |
| **Response**: Thanks for your comment.  In this paper, we adopt the performance metrics, including precision, recall and F1 to evaluate the bug-detecting performance. In particular, the precision, recall and F1 refer to the *overall* performance of bug detection in Figure 12. In Table III, we report the precision, recall and F1 in each category of VR/AR performance bugs. Therefore, we have essentially considered performance metrics for each category.  As for data split, we reported in Sec. IV-C. In particular, we split 80% of the dataset as a training set, 10% as a validation set, and 10% as a test set. More specifically, we split the data in each bug type based on cross-project codes to avoid similar code blocks existing in both training set and evaluation set.  In the revised paper, we supplemented additional details to clarify these issues. We excerpted the revised parts as follows:  In Section IV-C:  “Fig. 12 plots the overall performance-bug detection results of *PerfDectector* and the baseline models based on the constructed dataset.”  In Section IV-C:  “In particular, we split the data in each bug type based on cross-project codes to avoid similar code blocks existing in both training set and evaluation set.”  In Section IV-D:  “Table III presents the results of *PerfDectector* in detecting eight different types of performance bugs.” |
| Questions for authors’ response  -------------------------------  Questions:  1. How do you compare your approach with existing approaches on the technical part? |
| **Response**: Thanks for your question.  Our method distinguishes itself from traditional deep learning (DL)-based bug detector in the following perspectives. First, the proposed *PerfDetector* considers not only generic code characteristics (such as source code and AST) but also object invocations in its inputs so that it can address the unique challenges of VR/AR projects. Although object invocation or method invocation has been adopted for bug detection in previous research, our method originally includes *unique features* of VR/AR scenes. These features include virtual object-related invocations and usage, which do not exist in traditional apps (e.g., mobile apps). As a result, these three types of inputs construct the original semantic and syntactic information of VR/AR projects. Meanwhile, the proposed method emphasizes the unique features of VR/AR scenes. 2) Our *PerfDetector* specifically considers the programming language characteristics of VR/AR projects. Specifically, we adopt an LLM pre-trained on C# code (Code-T5), where C# has been the predominant language in Unity VR/AR projects. This design greatly enhances the detector's performance over other LLM-based methods. For a fair comparison, we have trained and evaluated off-the-shelf methods with the same hyperparameter setting based on our dataset in RQ2. |
| 2. What is the effectiveness of the semi-supervised labelling? |
| **Response**: Thanks for your question.  We admitted that manually labeling a large number of VR/AR performance bugs is extremely labor-intensive. To tackle this challenge, we proposed a semi-supervised method that leverages a small set of ground-truth labels to annotate the dataset first. Then, a data-labeling algorithm (Algorithm 2) was proposed to automatically label data samples.  Regarding your concern on the effectiveness of the proposed method, we have quantitatively evaluated the proposed semi-supervised learning approach from the following perspectives. 1) we randomly selected 825 samples (with identical probability) to verify clustering accuracy since the semi-supervised approach’s reliability depends on clustering accuracy. The selected samples have achieved 97.25% accuracy, thereby validating the reliability of our semi-supervised method. To clarify this issue, we provided more details in RQ1 and also reported it in Threats to Internal Validity (Section V). 2) it is worth mentioning that the first seven types of vulnerabilities in Table I were defined by previous paper [5]. The eighth type of bugs (COMPA) was defined by us. Specifically, we collected the commits by searching through compatibility-related keywords in the commit messages. To verify the correctness of the labeled data, we invited two well-experienced student volunteers with at least 1 year of Unity and C# development experience. Only if both volunteers agreed that the data was correct, we labeled the data as compatibility bug data. Thus, we get 635 COMPA bug data.  In the revised paper, we have supplemented these additional details. Please refer to the revised paper (the highlighted version) for more details. Hope your concerns will be eventually solved! |
| 3. Can you explain the details specified above on the evaluation process? |
| **Response**: Thanks for your question.  In this paper, we adopt the performance metrics, including precision, recall and F1 to evaluate the bug-detecting performance. In particular, the precision, recall and F1 refer to the *overall* performance of bug detection in Figure 12. In Table III, we report the precision, recall and F1 in each category of VR/AR performance bugs. Therefore, we have essentially considered performance metrics for each category. |
| After Rebuttal:  ------------------------------  I appreciate the additional evaluation provided by the authors, which resolved my concerns on results for each category and clustering. |
| **Response:** Thanks for your positive comment. |
| The distribution of positive / negative is more realistic now (and shows difference in results) but still does not reflect the reality as the same number of postives / negatives were used (there should be much more positive examples than negative examples). |
| **Response:** Thanks for your comment.  In order to justify the rationale of our data distribution, we have conducted two sets of experiments. In the first set of experiments, the amount of negative data is 1,500. This setting aims to make the model pay equal attention to each category of performance bugs. In the second set of experiments, the amount of negative data is equal to the total of all types of performance bugs. This distribution setting is more like training the model as a binary classification model. The experimental results show that our original data distribution has better results with a shorter training time due to the smaller dataset size. The supplemented results can be found in Table III. |
| The explanation of novelty does not help much as adding object invocation and support for C# are not really specific to the goal of the paper on VR performance bug detection, and it is also not clear whether training some existing approaches on the same dataset will outperform the proposed approach. |
| **Response**: Thanks for your comment.  We have explained the novelty of this paper in response to your second question. Please refer to the above response for more details.  Regarding your concerns about object invocation and C# language features, we further elaborate on the rationale of our design as the following aspects: i) our *PerfDetector* considers not only generic code characteristics (such as source code and AST) but also object invocations in its inputs so that it can address the unique challenges of VR/AR projects. Although object invocation or method invocation has been adopted for bug detection in previous research, our method originally includes *unique features* of VR/AR scenes. These features include virtual object-related invocations and usage, which do not exist in traditional apps (e.g., mobile apps). As a result, these three types of inputs construct the original semantic and syntactic information of VR/AR projects, meanwhile it emphasizes the unique features of VR/AR scenes. ii) Our *PerfDetector* specifically considers the programming language characteristics of VR/AR projects. Specifically, we adopt an LLM pre-trained on C# code (Code-T5), where C# has been the predominant language in Unity VR/AR projects. This design greatly enhances the detector's performance over other LLM-based methods. For a fair comparison, we have trained and evaluated off-the-shelf methods with the same hyperparameter setting based on our dataset in RQ2.  Regarding your concern on whether training existing approaches on the same dataset, we have essentially fine-tuned all those compared models on the same training set with the same hyperparameter settings for comparison fairness. We excerpted some details as follows.  In Section IV-C, “For a fair comparison, we fine-tune other LLM models based on our training set with the *same hyperparameter settings* and then evaluate their performance. Regarding the non-LLM (i.e., Word2vec), we also train the model based on our dataset with the *same hyperparameter settings*.” |
| Artifact assessment  3. Satisfactory, i.e., the artifacts are in line with what is declared in the submission form or the paper [OR] the authors explained why the artifacts are not provided and I find the explanation to be reasonable. |
| **Response**: Thanks for your positive evaluation of the artifact. |

**Point-by-Point Response to Reviewer 3**

|  |
| --- |
| **Paper summary** |
| This paper addresses performance bugs in VR/AR applications, which are critical for user immersion and interactivity. Due to the lack of labeled data and difficulties in understanding code structure, traditional methods struggle with bug detection. The authors collect 1,013 open-source VR/AR projects from GitHub and extract 10,950 buggy samples. These are categorized into eight types of performance bugs using a semi-supervised learning method, resulting in 8,520 labeled samples. Using the CodeT5 LLM, which learns from abstract syntax trees (ASTs), object invocation, and source code, the method shows superior performance in experiments, improving bug detection and enhancing VR/AR app reliability.  Strengths  ---------  + comprehensive dataset  + use of ML for bug detection  + rigorous evaluation  Weaknesses  ----------  - missing details in the methodology |
| **Response**: Thanks a lot for your positive comments on this paper. We also appreciate your constructive comments, which help us greatly improve the paper by conducting a major revision. We present the detailed responses to your concerns in a point-by-point manner as follows. |
| Novelty: This work is new work that compares PerfDetector with other state of the art methods.  Rigor: The authors collected a large dataset of VR/AR OSS projects. There is a systematic approach for categorizing bugs, a step that is needed for the labeling and detection approach. The authors work to overcome limitations of other LLMs by how they handle structural information. There is strong empirical evidence of the effectiveness of PerfDetector (i.e., recall, F1 score, etc.).  Relevance: This paper is relevant. First, VR/AR applications are representing a growing area within SE and the development of these systems and their challenges makes this work relevant. Second, these systems are highly interactive experiences and the performance bugs that impact the UX are therefore important to study. Third, the use of the state of the art ML techniques for beg detection is of interest to the community. Especially as the interest in applying AI and ML in enhancing the development process. |
| **Response**: Thank you for positive comments on our research work. |
| It is unclear how the COMPA bugs were manually labeled. (Methodology section C. Data labeling). |
| **Response**: Thanks for your comment.  Regarding the eighth type of performance bug data (COMPA), we further elaborate on the details of the labeling process as follows. Specifically, we collected the commits by searching through compatibility-related keywords in the commit messages. To verify the correctness of the labeled data, we invited two well-experienced student volunteers with at least 1 year of Unity and C# development experience. Only if both volunteers agreed that the data was correct, we labeled the data as compatibility bug data. Thus, we get 635 COMPA bug data.  In the revised paper, we have supplemented the additional details. We excerpted the major revisions as follows.  In Section III-C:  “Regarding “COMPA” bugs, we search in the commit messages of the collected dataset by using the keywords “compatibility” and “compat” (i.e., wildcard expression) and tag them as compatibility-related data. To ensure the correctness of labeling, we invite two well-experienced student volunteers with 1 year of Unity and C# development experience to verify the correctness of the labeling. Only if both volunteers agree that the labeling is correct, we confirm the data as compatibility bug data.”  In Section IV-A:  “Before applying the proposed data labeling based on semi-supervised learning, we define eight categories of performance bugs as specified in § III-B and obtain 1,421 initially labeled samples (including 635 manually labeled COMPA bug samples).” |
| The paper would be stronger with more details on the manual validation. The methodology is unclear and makes it difficult to replicate this work. |
| **Response**: Thanks for your comment.  To address the issue about the accuracy of label clustering, we randomly selected 825 samples (with identical probability) to verify clustering accuracy. The selected samples have achieved 97.25% accuracy, thereby validating the reliability of our semi-supervised method. To clarify this issue, we provided more details in RQ1 and also reported it in Threats to Internal Validity (Section V).  In the revised paper, we have supplemented more details on the methodology. We excerpt major revisions as follows.  In Section IV-C:  “To obtain a reliable result, we randomly select 825 samples and manually validate the accuracy of these labels generated by our proposed method. We find that these selected samples have achieved an average accuracy rate of 97.45%, demonstrating the reliability of the labeling results.”  In Section V:  “We have verified the accuracy of data labeling by randomly sampling partial data and the accuracy is acceptable (97.45%)” |
| Verifiability & transparency: There is an appendix online adding to the transparency and reproducibility of the study. However, there are many details missing in the methodology. For example, what was the small manual laying process. |
| **Response**: Thanks for your comment.  In the revised paper, we have supplemented additional technical details in the methodology from data collection, definition of VR/AR performance bugs, and data labeling to bug detection.  Regarding your concern about the small manual labeling process, we present the following details.  In Section III-C:  “We first obtain the small set of ground-truth labels based on the previous study [5]. These performance optimization commits are categorized into the first seven types as defined in § III-B by extracting the before-commit data. Then, the manually labeled seven categories of performance bugs (except for ``COMPA'' bugs) are obtained.” |
| Presentation: Figure 1 is well done. Figure 13 is difficult to read. It's interesting to have the color plus pattern bars, but in this figure it makes it bit more complicated to read, when printed, because of the figure's size. The paper is readable with minor grammar mistakes. |
| **Response**: Thanks for your comment.  In the revised paper, we convert Figure 13 into a table to further present the results. |
| Minor:  1. While the paper rightly emphasizes the critical nature of performance bugs in VR/AR apps due to their impact on user experience and potential to cause discomfort like dizziness, it overlooks that dizziness can also arise from other factors. |
| **Response**: Thanks for your comments.  We agree with you that user experience issues such as dizziness can be affected by different factors. However, this study investigates user experience issues mainly caused by code developing bugs in the software engineering (SE) perspective. As a result, our focus is to identify performance code bugs.  In the revised paper, we supplemented the details in Introduction (Section I). The revised text is shown as follows:  In Section I:  “The reason may refer to unreasonable development codes, such as the misuse of API.” |
| 2. The research questions moved earlier in the paper would improve readability and the motivation of this work. |
| **Response:** Thanks for your positive comment.  In the revised paper, we have adjusted the organization of this paper by presenting RQs in Introduction (Section I).  We excerpted the revised texts as follows:  In Section I:  “We focus on four research questions:  RQ1: What is the performance of the data labeling approach based on semi-supervised learning and LLM?  RQ2: What is the performance of the proposed *PerfDectector* compared with other LLM-based and conventional bug-detection methods?  RQ3: What is the performance of the proposed *PerfDectector* in detecting different types of performance bugs?  RQ4: What are the effects of *PerfDectector*’s different modules on bug-detection performance?” |
| Questions for authors’ response  1. Can you elaborate on how the semi-supervised learning approach based on BERT handles the diversity of VR/AR development SDKs and the heterogeneity of VR/AR devices in the data labeling process? |
| **Response:** Thanks for your question.  Your concern on handling the diversity of VR/AR SDKs and the heterogeneity of VR/AR devices can be addressed by our COMPA bugs. Regarding the eighth type of performance bug data (COMPA), we further elaborate on the details of the labeling process as follows. Specifically, we collected the commits by searching through compatibility-related keywords in the commit messages. To verify the correctness of the labeled data, we invited two well-experienced student volunteers with at least 1 year of Unity and C# development experience. Only if both volunteers agreed that the data was correct, we labeled the data as compatibility bug data. Thus, we get 635 COMPA bug data. |