Analyzing and Detecting Virtual/Augmented Reality Performance Issues with Large Language Models

ANONYMOUS AUTHOR(S)*

Virtual reality (VR) and augmented reality (AR) applications (apps) have received increasing attention recently. In contrast to conventional mobile apps, VR/AR apps have critical requirements for a user-immersive and interactive experience. The development life cycle of VR/AR apps is deeply influenced by software bugs, especially performance bugs, which may profoundly affect user experience. Despite the advent of recent large language models (LLMs) in bug detection, the lack of massive well-labeled data and the incapability of understanding the structural information of source code prevent the adoption of LLMs in detecting performance bugs of VR/AR apps. To comprehensively address these challenges, we propose a novel framework to construct a massive dataset with VR/AR performance bugs and detect them by a well-designed LLM-based method. We first collect 1,013 open-source VR/AR projects from GitHub and then extract 10,950 buggy samples at function levels. We then categorize the processed samples into eight types of performance bugs by a semi-supervised learning-based method according to the intrinsic features of bugs and obtain 8,520 labeled buggy samples. We then design a bug-detection method based on CodeT5 LLM, which can learn from abstract syntax tree (AST), object invocation, and source codes. The extensive experiments on the constructed dataset demonstrate the superior performance of the proposed method to conventional and LLM-based methods.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; • **Computing methodologies → Virtual reality**; **Mixed / augmented reality**.

Additional Key Words and Phrases: Bug Detection, Large Language Model, Performance Optimization

ACM Reference Format:
Anonymous Author(s). 2025. Analyzing and Detecting Virtual/Augmented Reality Performance Issues with Large Language Models. In *Proceedings of The ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '25).* ACM, New York, NY, USA, 21 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Virtual reality (VR) and augmented reality (AR) have become a hot topic recently [48]. The promising vision of the metaverse in 2021 has further boosted the development of diverse VR/AR systems. As reported by the International Data Corporation (IDC), the shipments of VR/AR headsets are expected to increase from 10.1 million in 2023 to 25 million units in 2026. The rapid proliferation of VR/AR systems has also boosted mobile VR/AR apps developed on top of software development kits (SDKs) and development frameworks.

As for VR/AR apps, *immersion* and *interactivity* are two of the most critical factors influencing users' experience [32]. Immersion refers to users' state of being thoroughly immersed in a simulated environment [3] so that they cannot almost distinguish the virtual world from the real one. Interactivity refers to the ability of a computer system to swiftly respond to users' actions (e.g., capturing a virtual ball or raising a virtual stick) [25]. Users' immersive experiences are often determined by the high fidelity of virtual 3D objects while interactive experiences are often affected

ISSTA '25, June 25–28, 2025, Trondheim, Norway

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

by the response time of VR/AR apps. Thus, users' excellent immersive and interactive experiences require not only high-performance computing facilities (e.g., CPU and GPU) but also the optimized usage of VR/AR SDKs and development frameworks [27].

Similar to conventional software, the development life cycle of VR/AR apps is also profoundly influenced by bugs. Differently, performance bugs become even more critical to VR/AR apps than conventional software applications due to users' high demand for immersive and interactive experiences in VR/AR environments. For example, performance bugs may greatly affect users' experience and cause uncomfortableness (e.g., dizziness) [27]. The reason may refer to unreasonable development codes, such as the misuse of API. It is laborious to identify, localize, and fix bugs in VR/AR apps due to the *diversity* of VR/AR development SDKs and the *heterogeneity* of VR/AR devices. Moreover, the rapid growth of VR/AR devices results in frequent updates of VR/AR apps to different versions, also leading to incompatible issues while incompatibility is often the root cause of performance bugs of VR/AR apps [4].

Although traditional bug detection methods can detect some bugs by static analysis [2, 11] and dynamic testing [30], they become struggling in detecting increasingly complex performance bugs in VR/AR apps. Two reasons result in the difficulty of traditional methods in detecting VR/AR performance bugs. First, VR/AR apps have been developed on top of game development engines, e.g., Unity game engine and Unreal Engine. As a result, VR apps have typically used complex and diverse APIs unlike conventional mobile apps only relying on simple SDKs. Thus, it is difficult for traditional methods to detect performance issues in VR/AR apps (more examples to be given in § 3.2). Second, in the VR/AR scenes, multiple virtual 3D objects can be invoked by one method (details to be given in Fig. 11 in § 3.4). Unfortunately, traditional bug detection methods cannot effectively detect these new yet complex bugs involving multiple virtual objects in VR/AR apps.

Recently, learning-based approaches have exhibited their advantages over traditional methods in detecting complex bugs driven by the rapid development of deep learning (DL) and Large Language Models (LLMs). These DL and LLM-based bug detection methods typically require massive well-labeled data (bug samples) to train DL models or LLMs. Unfortunately, a large volume of well-labeled performance-bug data in VR/AR domains is still largely missing, thereby restricting the adoption of the advanced DL and LLM approaches.

To overcome the above challenges in guaranteeing the software quality of proliferated VR/AR apps by bug detection, we propose a comprehensive framework (namely *PerfDectector*) to construct massive data of VR/AR performance bugs and detect them by an advanced LLM-based method. First, we collect 1,013 open-source repositories from GitHub according to relevance and popularity. We then define eight categories of performance bugs according to both the intrinsic features of VR performance bugs and a recent study [27]. To address the challenge of automatic labeling bug samples, we propose a novel data labeling approach based on semi-supervised learning and the BERT LLM [5] to embed the commit message, buggy file name, and function name. This method successfully categorizes 8,520 buggy samples into eight types. Existing LLMs mainly work on the sequences embedded by source codes while failing to handle structural information, which nevertheless is important for bug detection. To address this issue, we extract abstract syntax tree (AST) and object invocation, both of which have been serialized into sequences. We next feed AST sequences, object invocation sequences, and source codes into an LLM-based code encoder model – CodeT5 [46]. We evaluate the proposed *PerfDectector* over the constructed dataset by comparing conventional and LLM-based methods.

In summary, the main contributions of this work are summarized as follows.

- 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.

- After carefully defining eight types of performance bugs, we adopt semi-supervised learning based on BERT to embed buggy file names, buggy method names, and buggy commit messages for labeling eight categories of performance bugs.
- With the constructed dataset, 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 highly relevant to VR/AR app performance issues. *To the best of our knowledge, it is the first LLM-based tool to detect VR/AR performance bugs.*
- Extensive experiments on the constructed dataset verify the effectiveness of the proposed *PerfDectector*. Experimental results show that our *PerfDectector* outperforms existing methods (LLM-based methods and conventional DL method), e.g., achieving the highest accuracy (47.12), precision (50.99), recall (50.47), and F1 score (50.73). We make our approach available at https://anonymous.4open.science/r/PerfDetector-1B7D.

2 BACKGROUND

2.1 VR/AR App Development Frameworks

VR/AR applications have usually been developed on development frameworks [12], among which Unity is one of the most popular development frameworks [6]. Using C# as its main programming language, Unity provides strong cross-platform support, enabling developers to easily build applications on top of different VR or AR devices, such as Meta Quest, HTC Vive, and Hololens. Moreover, Unity also provides a VR/AR ecosystem for developers to access diverse resources (e.g., plugins and APIs). Specifically, Unity Asset Store offers a large number of VR/AR-related resources and plugins, including 3D models, sound effects, and scripts. In this way, developers can quickly build and extend their VR/AR applications. Moreover, many Original Equipment Manufacturers (OEMs) have provided developers with SDKs based on Unity to devise apps on specific devices. For example, OpenXR Mobile SDK [7] is provided by Meta to develop apps for Meta Quest and Oculus Rift. Similarly, ARCore is provided by Google to support AR app development on Google devices (e.g., Google Glasses) [29] while ARKit is released by Apple to develop AR apps on iOS [26]. With the provision of these development frameworks, app developers can invoke APIs from these SDKs to implement different functionalities, such as graphics rendering, object interaction, and hand tracking. Additionally, apps have often been updated periodically to add, remove, or upgrade modules to make the VR/AR systems adaptive to new changes although code changes may introduce performance compatibility bugs.

2.2 Performance Bugs of VR/AR Projects

As an important evaluation indicator of apps's quality, performance becomes extremely important for evaluating the execution effect and smoothness of VR/AR apps. It mainly consists of three metrics: graphics rendering, latency, and stability.

- **Graphics rendering** [34]: graphics rendering metrics mainly include frame rate [19] and graphic quality [43]. Frame rate refers to the number of image frames refreshed per second, usually expressed in Frames Per Second (FPS). A higher frame rate provides users with smoother visual effects and reduces motion sickness as well as dizziness. Graphics quality refers to the clarity and details of the images in VR/AR apps. High graphics quality can improve the user's perception of high fidelity and immersion. Frame rate and graphic quality are usually influenced by CPU and GPU resource consumption, etc. A high-quality VR/AR app needs to optimize the usage of these resources; otherwise, performance bugs in graphics rendering may occur.

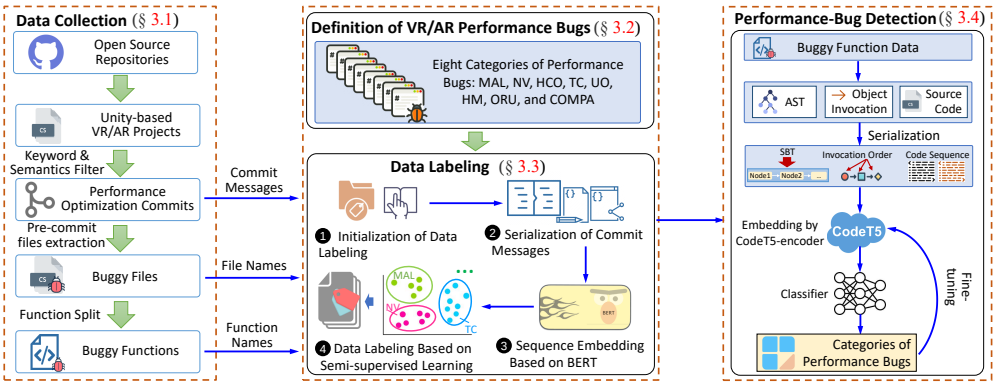

Fig. 1. Working Flow of the Proposed *PerfDectector*

- **Latency** [19]: Latency is the time difference between the user action and the response of a VR/AR app. A low latency is expected to improve the interaction experience of users. Being mainly determined by the response speed, latency is usually affected by multiple factors, such as the native code development of the app and the scheduling/utilization strategies of device resources. For example, a high latency may occur when the app's code contains inefficient and useless operations, or when there exist scheduling issues with threads (e.g., race condition). High latency is also the root cause of other performance bugs such as UI lags [4].
- **Stability** [18]: Stability requires that the application runs stably without crashing or suspending. For example, stability issues may happen when a vulnerability in the thread-scheduling task leads to lock contention. When VR/AR apps perform poorly in terms of low stability, performance bugs such as UI glitches and black screens may occur [4, 24].

New functionalities have typically been added/upgraded by VR/AR app developers when updating the app version. Adding new features to the app without careful testing may bring side effects, such as performance bugs, which nevertheless are often overlooked by developers and not discovered by users even after the app is publicly launched. Therefore, how to automatically and swiftly detect performance bugs in VR/AR apps is an important issue, which is the aim of this paper.

3 METHODOLOGY

Fig. 1 depicts the working flow of the proposed performance-bug detector (*PerfDectector*). Firstly, we collect Unity-based VR/AR projects (i.e., commits) from the open source repositories (in § 3.1). Then, we formally define eight categories of representative performance bugs in VR/AR apps (in § 3.2). We next present a semi-supervised learning-based LLM method to label the collected VR/AR commits (in § 3.3). Finally, we design an LLM-based method to train the model based on the labeled data (in § 3.4).

3.1 Data Collection

We collect Unity-based VR/AR open-source projects mainly from GitHub due to its highest popularity. Firstly, we search for performance-optimization-related commits using a wide range of keywords with semantic features relevant to performance issues of VR/AR apps. Specifically, we collect Unity-based VR/AR projects by using keywords, such as "Virtual Reality", "VR", "VR Unity", "AR", "AR Unity" and "Augmented Reality" because Unity is one of the most popular developing frameworks for VR/AR apps (over 63% market share [41]). We mainly focus on C# language since the Unity framework mainly uses C#. We then select the projects based on the number of *stars* (i.e., more stars mean more popular of the projects). Next, we adopt AST analysis and `diff` tools to split

Algorithm 1: Extract Buggy Function

Input: M: The source code of the buggy file; s: Start line of the buggy code snippet;
e: End line of the buggy code snippet.
Output: B: The source code of the buggy function;

```
function extract_buggy_function(M,s,e)
method_root_node ← ∅
nearest_struct_node ← ∅
B ← ∅
//Generate the AST of the buggy file
ASTtree_set = generate_AST(M)
line_diff = +∞
//Find the nearest structure node (method root node)
for node in ASTtree_set do
    if node.type == Type_declarationContext then
        if node.startline ⩽ s and node.endline ⩾ e then
            if s-node.startline <line_diff then
                line_diff = s-node.startline
                nearest_struct_node = node
            end
        end
    end
end
if type(nearest_struct_node) = method_declaration_node then
    method_root_node = nearest_struct_node
    B ← M[method_root_node.startline:method_root_node.endline]
    return B
else
    return ∅
end
```

functions and extract buggy functions. Finally, we obtain raw commit data of VR/AR projects. We elaborate on the data collection procedure as follows.

When collecting Unity-based VR/AR projects, we mainly utilize `commit` property in GitHub to collect performance bugs, where a commit in a GitHub project is a single snapshot reflecting the project's staged changes. Each commit contains a set of changes to files, along with the developer's description of those changes (i.e., commit message). We mainly focus on the commits related to performance optimization in VR/AR apps. We use prefixes of keywords, such as "fix", "bug", "patch", "resolve", "performance", "speed up", "accelerat" (representing "accelerate", "accelerating", and "acceleration"), "fast", "slow", "latency", "contention", "optimiz" (representing "optimize", "optimizing", and "optimization"), and "efficien" (representing "efficient" and "efficiency") to search in and collect commit messages in each project.

Based on these collected commit messages, we use an adaptive LLM model – BERT to embed both collected commit messages and uncollected messages, thereby obtaining the feature vectors. Based on the Transformer encoder [42] architecture, BERT was proposed by Google as a pre-trained deep learning model for natural language understanding tasks [5]. It consists of 12 Transformer encoders attached behind the embedding layer and the positional encoder layer. We compare the uncollected commit messages with the collected commit messages in terms of the feature vectors. If the feature vector of uncollected commit messages is close enough to the nearest feature vector of collected commit messages, these uncollected commit messages will be included in the collected commit messages. Specifically, we measure the closeness of the uncollected sample's feature vector

V and the collected sample's feature vector W by the Euclidean distance as follows,

$$d(V, W) = \sqrt{\sum_{k=1}^{n} (v_k - w_k)^2}, \tag{1}$$

where v_k and w_k are values of the two feature vectors V and W in the k^{th} ($k \in [1, 2, \cdots, n]$) dimension, respectively.

In this way, we attain the corresponding performance-optimization-related commits. Since each commit contains a set of changes on code files, we then extract the `.cs` files before committing, called *pre-commit files*, thereby obtaining the buggy files. For each buggy file, we locate the code-changed lines, which are called the *buggy code snippet*. We next further split the buggy functions according to the buggy code snippet. Algorithm 1 describes this process in detail. Firstly, We adopt ANTLR4[1], which is a syntax parser to generate abstract syntax trees (ASTs) from source codes to generate a class-level AST (i.e., generate_AST in line 6). Then, we traverse the AST and find the nearest structure node (i.e., the smallest line_diff) before the start line of the buggy code snippet. A structural node is a node featured by Type_declarationContext as the node type. It is usually regarded as the root node of a method or variable declaration structure. We only keep the structural nodes related to method declarations. By obtaining the start line and end line of the structure node, we can get the range of code lines at the source code level, thereby accurately locating the range of the buggy function.

3.2 Definition of VR/AR Performance Bugs

After collecting performance-related buggy functions, we characterize performance bugs in detail. Referring to the dataset proposed by the VR performance optimization research [27] and our collected dataset from GitHub, we characterize major VR/AR performance bugs and categorize them into eight categories, which are shown as follows.

3.2.1 Misusage of API and Language (MAL). MAL refers to the usage of inefficient API or misuse of API functionality, but also includes inefficient usage of data types due to lack of familiarity with programming language features, thereby incurring a slow program and impacting VR/AR performance. Fig. 2 shows an example excerpted from our collected Github VR/AR projects. `Dequeue()` and `Enqueue()` methods in line 11 and line 13, respectively, are adopted to remove and add items of `stabilitySamples`, indicating that `stabilitySamples` is a queue. However, it is inefficient to iteratively invoke 'elementat' queue methods compared with calling list methods, implying an MAL bug in method `AddGazeSample()`.

```
1  private void AddGazeSample(Vector3 positionSample, Vector3 directionSample)
2  {
3      GazeSample newStabilitySample;
4      newStabilitySample.position = positionSample;
5      newStabilitySample.direction = directionSample;
6      newStabilitySample.timestamp = Time.time;
7      if (stabilitySamples != null)
8      {
9          if (stabilitySamples.Count >= StoredStabilitySamples)
10         {
11             stabilitySamples.Dequeue();
12         }
13         stabilitySamples.Enqueue(newStabilitySample);
14     }
15 }
```

Fig. 2. Misusage of API and Language (MAL)

[1] https://github.com/antlr/antlr4

```
private void SpawnLight()
{
    var light = (GameObject)Instantiate(Resources.Load("Light"));
    // random duration and easing
    var duration = Random.Range(0.5f, 4f);
}
```

Fig. 3. Negative Visualization (NV)

3.2.2 Negative Visualization (NV). NV refers to the fact that the inappropriate usage of graphics rendering functions, object material texture, and scene settings within a VR/AR app affects the loading speed and smoothness of the program, thereby affecting the user's visual experience. For example, NV may cause a black screen or a lagging screen display, consequently degrading the user's immersive experience. Fig. 3 shows an example. In the method `SpawnLight()`, the process to guide the light to the edge is missing, thereby affecting the visual effect, indicating an NV bug.

3.2.3 High-Cost Operations (HCOs). HCOs often incur unnecessary consumption of computational resources due to improper code design, thereby degrading the performance of the VR/AR app on the device and causing a lag in the app. For example, the absence of a cached object can result in high costs when reading and writing scene objects in VR/AR apps. Fig. 4 shows an example. In line 6, `Camera.main` is wrongly used to construct a ray from the current touch coordinates, thereby causing unnecessary high reading and writing costs. This inefficient method can be replaced by caching object `CameraCache` with `CameraCache.Main` being constructed.

```
protected virtual void Update()
{
    foreach (Touch touch in Input.touches)
    {
        // Construct a ray from the current touch coordinates
        Ray ray = Camera.main.ScreenPointToRay(touch.position);

        switch (touch.phase)
        {
            case TouchPhase.Began:
            ...
        }
    }
}
```

Fig. 4. High-cost Operations (HCOs)

3.2.4 Thread Conflict (TC). TC refers to network conflict or resource access conflict (e.g., lock contention in race conditions) caused by improper thread allocation or task ordering in tasks, such as synchronization. The TC bugs can cause the app to lag or even crash. Fig. 5 shows an example of TC. Due to the lack of appropriate asynchronous operation configuration, the method `ContainsAsync()` is executed without releasing the main thread, thereby potentially causing a TC.

```
public Task<bool> ContainsAsync(int key)
{
    return Context.ZoneEntries.AnyAsync(z => z.ZoneId == key);
}
```

Fig. 5. Thread Conflict (TC)

3.2.5 Useless Operations (UOs). Rather than substantially enhancing the app's functionality, UOs reduce the app's responsiveness. Typical examples include loops that do not stop in time and method invocations to useless virtual objects in scenes. In Fig. 6, the `for` loop in line 5 is always executed in its entirety, thereby incurring a waste of time. To address this, a counter can be placed inside to stop the loop earlier.

```
public IEnumerable<ListTreeNode<T>> Children
{
    get
    {
        for (int i = ValueIndex; i < m_Values.Count; ++i)
        {
            if (m_Values[i].ParentIndex == ValueIndex)
            {
              yield return new ListTreeNode<T>(m_Values, i);
            }
        }
    }
}
```

Fig. 6. Useless Operations (UOs)

```
protected virtual void UpdateRenderData()
{
    ...
    GameObject[] unusedGameObjects =
    {
        origin.validObject,
        origin.invalidObject,
        repeatedSegment.validObject,
        repeatedSegment.invalidObject,
        destination.validObject,
        destination.invalidObject
    };
    foreach (GameObject unusedGameObject in unusedGameObjects)
    {
        if (unusedGameObject != null
            && (unusedGameObject != pointsData.start
            && unusedGameObject != pointsData.repeatedSegment
            && unusedGameObject != pointsData.end
            || unusedGameObject == repeatedSegment.validObject
            || unusedGameObject == repeatedSegment.invalidObject))
        {
            unusedGameObject.SetActive(false);
        }
    }
    ...
}
```

Fig. 7. Heap Misallocation (HM)

3.2.6 Heap Misallocation (HM). HM refers to the misusage of heap allocation. For example, when garbage collection is not precisely controlled and intervened, it can lead to additional performance overhead. Moreover, the usage of inefficient data structure (e.g., array) can also cause HM. As shown in Fig. 7, a temporary array of objects is wrongly created and allocated, thereby resulting in HM and degrading the performance.

3.2.7 Overcomplex Rendering and UI (ORU). ORU refers to the overcomplex scene or UI design, which causes the app to render erroneously or slowly when the scene or UI changes. As a result, the app lags or even crashes. Fig. 8 shows an example, in which the overcomplex usage of parameters `camera.pixelWidth` and `camera.pixelHeight` in constructing a `RenderTexture` object in line 9 causes an ORU bug.

3.2.8 Compatibility (COMPA). The COMPA bug is a new type of performance bug found in our constructed dataset. These bugs have received extensive attention recently in many developer and user communities. Depending on hardware and software diversity, the COMPA bugs can be categorized into hardware COMPA bugs and software COMPA bugs. Hardware COMPA bugs

```
protected void OnEnable()
{
    camera = GetComponent<Camera>();
    tempSecondaryCamera = new GameObject().AddComponent<Camera>();
    tempSecondaryCamera.enabled = false;
    Debug.Assert(replacement);
    mat = new Material(shader);
    Debug.Assert(mat);
    TempRT = new RenderTexture(camera.pixelWidth, camera.pixelHeight, 0);
}
```

Fig. 8. Overcomplex Rendering and UI (ORU)

often accompany performance degradation due to compatibility issues when the app is running on a VR/AR device or interacting with peripheral devices, such as controllers. Take one of the most popular social VR apps – *RecRoom*, as an example, in which a device compatibility issue was found on Oculus Quest 1 (an earlier VR device than Oculus Quest 2 and Oculus Quest 3)[2]. Software COMPA bugs are caused by compatibility issues between diverse versions of mobile operating systems and development frameworks. For example, a degraded performance was found when maintaining the app compatible with a new version of the Unity development framework[3]. Considering the code segment in Fig. 9, a compile error appears in line 1 when updating the app to a new version of Unity, indicating a COMPA bug.

```
public static LiteCoroutine StartCoroutine(IEnumerator routine, bool runImmediate = true, bool wakeSystem = true)
{
    if (!Initialize()) { return null; }
    var handle = defaultManager.StartCoroutine(routine, runImmediate);
    if (!handle.IsDone && wakeSystem) { WakeUp(); }
    return handle;
}
```

Fig. 9. Compatibility (COMPA)

3.3 Data Labeling

After defining eight categories of VR/AR performance bugs, it is necessary to develop an automated approach to help developers detect bugs in VR/AR apps. However, it is non-trivial to achieve this goal due to the small number of available bug labels, which nevertheless are important to train a supervised bug detector. It is extremely laborious to manually label a large number of performance bugs from our collected dataset. To address this challenge, we adopt a *semi-supervised learning* method based on recent advent LLMs to learn from the small set of available ground-truth bugs, as shown in Fig. 1. Benefiting from the strong representation capability of LLMs, our proposed semi-supervised LLM approach can successfully label performance bugs into eight categories. We further elaborate on the detailed procedure as follows.

❶ *Initialization of Data Labeling*. We first obtain the small set of ground-truth labels based on the previous study [27]. These performance optimization commits are categorized into the first seven types as defined in § 3.2 by extracting the before-commit data. Then, the manually labeled seven categories of performance bugs (except for "COMPA" bugs) are obtained. 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

[2]https://recroom.zendesk.com/hc/en-us/articles/7901926792343-Device-Compatibility
[3]https://forum.unity.com/threads/is-anyone-else-getting-terrible-vr-performance-with-2022-lts-fixed.1446334/

Algorithm 2: Data Labeling

Input: $\mathbf{L}_i$: Feature vector set of initially labeled sample of i^{th} performance bug type;
U: Feature vector set of unlabeled sample; n: Bug type amount; t: Distance threshold
Output: **B**: Labeled dataset;

```
1  B ← ∅
2  //Find centers of 8 performance bug types
3  for i := 1; i ⩽ n; i:= i + 1 do
4  |   C_i = center(L_i)
5  end
6  //Labeling unlabeled data
7  for U in U do
8  |   distance_set ← ∅
9  |   for i := 1; i ⩽ n; i:= i + 1 do
10 |   |   d(U, C_i) = Euclidean(U, C_i)
11 |   |   distance_set ← d(U, C_i)
12 |   end
13 |   min_distance = min(distance_set)
14 |   if min_distance < t then
15 |   |   U[label_index] = distance_set.index(min_distance)
16 |   |   B ← U
17 |   end
18 end
19 return B
```

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.

❷ *Serialization of Commit Messages.* Since the commit message contains bug-fixing information, it can capture the key characteristics of the bug categories. Therefore, we construct a sequence represented by "`[File Name]` + `[Function Name]` + [commit message]", for each labeled sample and unlabeled sample. For samples where a single commit message contains more than two function modifications, we filter out the buggy function samples belonging to this type of commit message, since these samples may introduce additional labeling noise due to modifications irrelevant to performance bugs.

❸ *Sequence Embedding Based on BERT.* After getting the sequence of each sample, we adopt the advanced LLM – BERT to embed the sequence and its output is the feature vector of the sequence.

❹ *Data Labeling Based on Semi-supervised Learning.* We next design a data-labeling algorithm based on semi-supervised learning, as shown in Algorithm 2. Among the labeled samples, we compute the feature vector's centroid for the labeled samples of each category of performance bugs. We thereby get eight centers of feature vectors (line 2-line 5). The process is depicted as follows. Specifically, the feature vector's centroid of the i^{th} category of performance bugs is denoted by $\mathbf{C}_i$ ($i \in [1, 2, \cdots, 8]$), which is calculated by

$$\mathbf{C}_i = \frac{1}{m} \sum_{j=1}^{m} \mathbf{X}_{ij}, \tag{2}$$

where m is the size of labeled samples of the i^{th} category of performance bugs and $\mathbf{X}_{ij}$ represents the j^{th} labeled sample in the i^{th} category. The k^{th} dimension value of the i^{th} center feature vector c_{ik} is computed as follows,

$$c_{ik} = \frac{1}{m} \sum_{j=1}^{m} x_{ik(j)}, \tag{3}$$

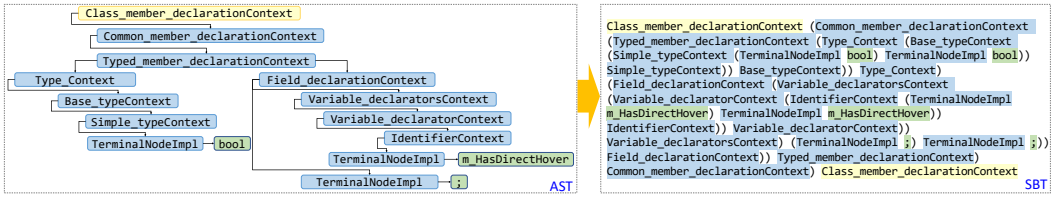

Fig. 10. SBT Process

where $x_{ik(j)}$ represents the k^{th} dimension value of the j^{th} labeled sample in the i^{th} category.

For each unlabeled sample U, we compute its Euclidean distance from each feature vector's centroid (line 10) by

$$d(U, C_i) = \sqrt{\sum_{k=1}^{n} (u_k - c_{ik})^2} \tag{4}$$

where U is the feature vector of the unlabeled sample and u_k is the k^{th} dimension value of the unlabeled feature vector. We find the feature vector's closest centroid, which has the smallest Euclidean distance (line 13). If the Euclidean distance is less than the given threshold t, we include the corresponding type of the performance bug in the labeled dataset $\mathbf{B}$.

3.4 Performance Bug Detection

After getting the labeled buggy function dataset, we then use this constructed dataset to train the proposed learning-based automatic method called *PerfDectector* for detecting performance bugs. To let *PerfDectector* learn bug features comprehensively, we utilize three features of the buggy functions: AST, object invocation, and source code, as shown in Fig. 1. After that, we adopt another advent LLM model – CodeT5 to classify bugs. It is worth mentioning that this CodeT5 LLM can be further fine-tuned. The detailed design is elaborated as follows.

3.4.1 AST. As an abstract representation of the syntactic structure of source code, AST has been commonly used in programs such as compilers and interpreters to represent the syntactic structure of program code. It is represented by a tree structure, where each node has *type* and *value* information. The *type* information in each node indicates the type of the syntactic structure of the node, such as "*if statement*", "*function call*", etc. The *value* information indicates the specific value of the node, such as the variable name and literal value. It is difficult to directly let an LLM learn from AST. To address this issue, we exploit an algorithm called Structure-based Traversal (SBT) proposed by [14] to traverse AST and obtain the AST sequence while preserving its structural information. SBT uses matched parentheses to record the structure information, being represented as "`(node information) node information`". The information about the node's children is placed after the node information in parentheses in the order of preorder traversal. We only preserve the *type* information for non-leaf nodes while keeping the *type* and *value* information for the leaf nodes. Fig. 10 shows an example of converting an AST to an SBT.

3.4.2 Object Invocation. In VR/AR programs, "object invocation" usually refers to the process of manipulating objects or interacting with objects in the VR environment. It is usually implemented using a variety of APIs [28, 39]. Although object invocation or method invocation has been adopted for bug detection in previous research [35, 45], 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 [34]. `RenderTexture`-related APIs are used to create a life-like vision on a screen so as to enable immersive

```
protected override void OnDisable()
{
    base.OnDisable();
    if (initaliseListeners != null)
        StopCoroutine(initaliseListeners);
    if (setDestination != null)
        StopCoroutine(setDestination);
    ManageDestinationMarkers(false);
    if (createdCollider)
        Destroy(pointCollider);
    if (createdRigidbody)
        Destroy(pointRigidbody);
}
```

Fig. 11. Object Invocation

experiences. The object invocations involve several aspects, such as scene element invocation, user input invocation, device camera and viewpoint invocation, animation invocation, virtual physics engine invocation, audio invocation, network invocation, interactive object invocation, etc. Fig. 11 shows an example, in which the method of `OnDisable()` invokes multiple objects, such as `initaliseListeners`, `setDestination`, `createdCollider`, `pointCollider`, `createdRigidbody`, and `pointRigidbody`. 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. We extract these objects by searching the AST for a node whose *type* information is `Method_invocation`, and whose *value* information is the specific content of the invocation object. Since a method may call more than one object, we convert the objects to the sequential form of `call_1 object_1, call_2 object_2,...` as the model input.

3.4.3 Source Code. Source code refers to the original program text written by developers. It includes buggy functions' basic semantics. In order to let an LLM extract features from source codes, we also traverse the source code to a sequence.

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. After obtaining the above three types of sequences, we adopt a CodeT5-based model for automatic performance-bug detection. As an encoder-decoder LLM for software engineering, CodeT5 was designed on top of a pre-trained language model – Text-to-Text Transfer Transformer (T5) model [33] proposed by Google [42]. Compared with the conventional Transformer, it removed the Layer Normalization bias, placed the Layer Normalization outside the residual path, and used a different position embedding. CodeT5 utilizes different pre-training corpus including eight programming languages. We adopt this model since it also supports C# (used for most Unity projects). In order to get the feature vectors of the three sequences, we adopt the CodeT5 encoders to embed the aforementioned three sequences and attach a neural network as a classifier after the encoders. In the training phase, we use the training set of the labeled samples to fine-tune the encoder and train the classifier. In the validation and test phase, we use the validation set and test set to check the effectiveness of performance bug detection of *PerfDectector*.

4 EXPERIMENTS AND EVALUATION

This section presents experimental results to evaluate the performance of *PerfDectector* in detecting VR/AR performance bugs and analyze the influencing factors of its performance. We aim to answer the following research questions.

Table 1. Distribution of Initially Labeled Samples

Performance Bug Category	Initial # of samples
MAL	198
NV	49
HCOs	101
TC	21
UOs	132
HM	283
ORU	2
COMPA	635
Total	1,421

Table 2. Overall Data Distribution

Performance Bug Category	# of samples
MAL	747
NV	1,218
HCOs	1,631
TC	24
UOs	1,484
HM	1,771
ORU	82
COMPA	1,563
Negative Data	1,500
Total	10,020

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?

4.1 Data Preparation & Experiment Configurations

As mentioned in § 3.1, we collect 1,013 popular VR/AR projects from GitHub. After the filtering process based on well-defined keywords and semantics, we obtained 28,950 commits related to the performance optimization of VR/AR apps. We next extract the buggy functions from the pre-committed C# files. Thereafter, we obtain 9,529 function-level buggy samples. Before applying the proposed data labeling based on semi-supervised learning, we define eight categories of performance bugs as specified in § 3.2 and obtain 1,421 initially labeled samples (including 635 manually labeled COMPA bug samples). Table 1 presents the distribution of the initially label samples according to eight categories. At last, 10,950 buggy samples are obtained in total.

The model hyperparameters are configured as follows. The batch size is configured as eight and the maximum number of epochs is chosen to be 100. We invoke and train LLMs by using APIs and models provided by HuggingFace [47], which is a community platform providing open-source NLP models and tools. As for the optimizer, we utilize Adam [15] with the learning rate 10^{-5}. The dropout rate is chosen to be 0.1. All experiments are conducted on a server configured with the GPU of NVIDIA GeForce RTX 3090.

4.2 RQ1: Performance of proposed data labeling approach

As mentioned in § 3.3, we propose a data labeling approach based on semi-supervised learning to automatically label the collected data samples. The average threshold t of Euclidean distance is chosen to be 10. When the distance between the feature vector of an unlabeled sample and the centroid of any of the eight feature vectors is less than the threshold, this sample will be labeled as the known category. We also filter out those commit messages not belonging to any type of performance bugs. At last, we categorize 8,520 buggy samples into eight types. Notably, unbalanced data samples also exist in our collected VR/AR projects. To address this problem, we also collect 1,500 functions from the latest version of each collected VR/AR project as functions without bugs since previous bugs should be fixed in these functions (i.e., no such bugs in the latest version). Then, we manually check them. These newly collected functions are supplemented to our dataset as manually negative data.

Table 2 lists the distribution of all the processed samples according to eight types of performance bugs. It can be found that HM is the type with the largest number of samples, accounting for

20.78% of the buggy data. Moreover, HCOs and COMPA also account for 19.14% and 18.35% of the buggy data, respectively. This implies that these three types of bugs are the most frequent performance bugs in existing VR/AR projects. 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.

Answer to RQ1: HM, HCOs, and COMPA are the three most popular performance bugs in VR/AR projects. Manually validated results verify the reliability of data labeling.

4.3 RQ2: Performance of *PerfDectector* compared with other LLM-based and conventional bug detection methods

To verify the performance of *PerfDectector*, we compare the proposed *PerfDectector* and other four state-of-the-art approaches based on LLMs and conventional neural networks. We consider accuracy, precision, recall, and F1-score as the evaluation metrics. We adopt representative LLMs including *RoBERTa* [20], *CodeBERT* [8], and *GraphCodeBERT* [10].

- ***RoBERTa***: *RoBERTa* is a natural language pre-trained model based on BERT. *RoBERTa* optimizes the training process and data processing so as to attain better performance than BERT. When using the model, we directly migrate pre-trained models based on natural language tasks of code understanding and bug detection.
- ***CodeBERT***: *CodeBERT* is designed for tasks at the transformation of programming languages and natural language. It is pre-trained on a large number of codes and natural language texts with the support of six kinds of programming languages, demonstrating excellent performance on code search, code document generation, and code-natural language crossover tasks.
- ***GraphCodeBERT***: *GraphCodeBERT* is an extension of *CodeBERT* by considering the structural information of the code in the pre-training phase. The model combines the code text, AST information, and variable dependence information as the model input. *GraphCodeBERT* shows better performance than *CodeBERT* on several code-understanding and code-generation tasks.

We utilize these LLMs separately to encode the inputs and attach the classifiers afterward. We chose these LLMs mainly because they are the most popular LLMs in software engineering research and are widely used to capture code semantic information [13]. In particular, neither these LLMs nor the CodeT5 encoder included in *PerfDectector* has been fine-tuned by pre-training data and pre-training tasks, which are relevant to VR/AR performance bug detection.

As for the conventional deep-learning-based approaches, we adopt *Word2vec* [23], which has been widely adopted in bug detection tasks, such as [1, 38]. Since Word2vec directly maps the inputs to vectors in a high-dimensional space, we obtain feature vectors of source codes, AST, and object invocations. We then attach a classifier for performance bug detection similar to the proposed *PerfDectector*. For conventional settings, we split 80% of the dataset as a training set, 10% as a validation set, and 10% as a test set. In particular, we split the data in each bug type of cross-project codes to avoid similar code blocks falling in both the training set and evaluation set. 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*.

Fig. 12 plots the overall performance-bug detection results of *PerfDectector* and the baseline models based on the constructed dataset. It can be found that the proposed *PerfDectector* outperforms other compared approaches. In particular, *PerfDectector* achieves the highest accuracy (47.12), precision (50.99), recall (50.47), and F1 score (50.73). Moreover, all LLM-based approaches (including

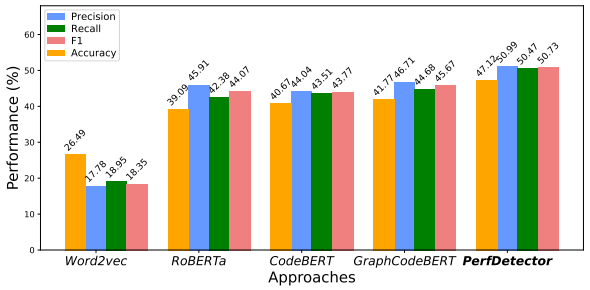

Fig. 12. Performance-bug detection results of compared approaches

our *PerfDectector*) perform better than *Word2vec* in terms of accuracy, precision, recall, and F1 score, indicating the effectiveness of LLMs in detecting performance bugs.

Among LLM-based methods including *RoBERTa*, *CodeBERT*, *GraphCodeBERT* and *PerfDectector*, the proposed *PerfDectector* is 15.11% better than *CodeBERT*, 15.90% better than *CodeBERT* and 11.08% better than *GraphCodeBERT* in terms of F1 score. This performance improvement may be attributed to (i) the adopted CodeT5 encoder and (ii) the newly-constructed pre-training corpus (based on Unity VR/AR projects) for CodeT5, which supports the C# language. This result implies that our approach has a good understanding of the code semantics of VR/AR projects, thereby achieving excellent performance in detecting performance bugs.

Answer to RQ2: *PerfDectector* outperforms baseline models, demonstrating its effectiveness in detecting performance bugs.

4.4 RQ3: *PerfDectector*'s performance in detecting different types of performance bugs.

We further evaluate the performance of *PerfDectector* in detecting different types of performance bugs. Table 3 (left half) presents the results of *PerfDectector* in detecting eight different types of performance bugs. In particular, *PerfDectector* can be regarded as an effective tool to detect whether a function contains any performance bugs since its F1 score in *negative data* reaches 100.00%. Besides *negative data*, *PerfDectector* also achieves a 100.00% F1 score in detecting TC bugs. This result implies that TC bugs have quite salient syntactic and semantic features so that they can be easily identified. TC bugs often appear in codes related to sessions and thread calls. Differently, *PerfDectector* only obtains 26.67% and 29.30% in detecting ORU bugs and UOs bugs, respectively. This inferior performance may be ascribed to relatively complex code structures. How to improve the performance of *PerfDectector* or other LLM-based methods will be a future direction.

Notably, we focus on constructing a multi-class classification model in our detector. Thus, we aim to balance the amount of negative data and the amount of data in each positive class as much as possible to make them close to a realistic distribution. Considering the laborious process of manually

Table 3. *PerfDectector*'s performance in detecting different types of bugs

Performance Bug Category	Original Data			Balanced NP Data		
	Precision (%)	Recall (%)	F1 (%)	Precision (%)	Recall (%)	F1 (%)
MAL	32.08	45.33	37.57	37.84	37.33	37.58
NV	42.02	40.98	41.49	38.64	27.87	32.38
HCOs	45.61	31.71	37.41	39.80	47.56	43.33
TC	100.00	100.00	100.00	100.00	100.00	100.00
UOs	32.26	26.85	29.30	27.89	27.52	27.70
HM	35.33	29.78	32.32	34.78	26.97	30.38
ORU	33.33	22.22	26.67	50.00	11.11	18.18
COMPA	38.30	57.32	45.92	36.54	48.41	41.64
Negative Data	100.00	100.00	100.00	99.88	100.00	99.94
Overall	50.99	50.47	50.73	51.71	47.42	49.47

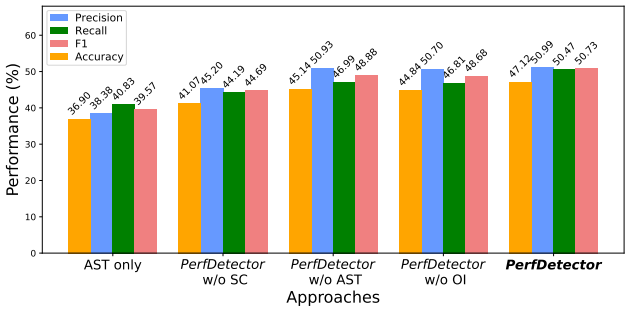

Fig. 13. Ablation study with removal of key components of *PerfDectector*

labeling negative data, we cannot label all negative data due to the huge data amount. Recent study [37] indicates that the amount of positive data is close to that of negative data. Therefore, in order to validate the rationality of the dataset distribution settings, we supplement a new experiment by adding an amount of negative data so as to be consistent with the amount of positive data of all types. We name this additive dataset as *Balanced Negatives and Positives* (Balanced NP) dataset. The results are reported in the right half of Table 3. We observe that *PerfDectector*'s F1 value in Balanced NP is not better than *PerfDectector* in the original dataset (i.e., 2.55% less than *PerfDectector* in F1). Moreover, *PerfDectector* in the original dataset also has higher F1 values than that in Balanced NP in detecting most types of performance bugs. This result also implies that our *PerfDectector* achieves better performance in the original dataset than that in Balanced NP with less training time (because of the smaller amount of data).

Answer to RQ3: *PerfDectector* performs the best in detecting TC bugs, but performs poorly in detecting ORU and UOs bugs. The data distribution setting for *PerfDectector* is reasonable.

4.5 RQ4: Effects of different modules on detection performance

As mentioned in § 3.4, we adopt AST, object invocation, and source code as the model inputs. They may affect the detection performance of *PerfDectector*. Thus, we conduct an ablation study to quantify the contributions of these three inputs to the performance. We use "*AST only*", "*PerfDectector without Source Code (SC)*", "*PerfDectector without AST*", "*PerfDectector without (w/o) 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, 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. Therefore, three input sequences all contribute to the performance improvement of the proposed *PerfDectector*.

Answer to RQ4: Three types of input sequences of *PerfDectector* all contribute to the optimized performance. Source code information contributes the most among them.

4.6 Discussion: Incorrect Detection Results

As mentioned in § 4.4, we find that *PerfDectector* performs worse in detecting ORU and UOs types of bugs than other types. We further investigate the root causes behind these results. Specifically, we select two examples of incorrect detection results belonging to ORU and UOs types of bugs. As shown in Fig. 14, a UO bug exists but *PerfDectector* mistakenly categorizes it as HM. This may be because the usage of the list datatype in the method disturbs the model's discrimination since the list datatype often relates to heap allocation (e.g., `Add()`). Moreover, multiple `if` structures make it difficult for the model to understand the complex structure and semantics of the code. Fig. 15 shows another example of an ORU bug existing while *PerfDectector* wrongly detects it as a COMPA bug. The incorrect detection may be attributed to complex `if` structures. In addition, the semantics of "update of the SDK" may mislead the model to categorize this bug as a COMPA bug. In summary, complex syntax structures, the invocation of the methods related to other bugs, and misleading keywords may impact *PerfDectector*'s detection accuracy.

```
[RequiresBehaviourState]
public virtual void Add(CollisionNotifier.EventData collisionData)
{
    if (Elements.Contains(collisionData) || !IsValidCollision(collisionData))
    {
        return;
    }
    Elements.Add(CloneEventData(collisionData));
    Added?.Invoke(collisionData);
    ...
}
```

Fig. 14. Case 1: Incorrect detection of UOs bug

5 LIMITATION AND THREATS TO VALIDITY

Limitation. The major limitation of our *PerfDectector* lies in the detection granularity of performance bugs. The current version of *PerfDectector* mainly focuses on detecting buggy functions rather than pinpointing the exact lines where the performance bug is located. We will explore a more precise bug-location method (e.g., line-level bug localizer) to help developers fix discovered bugs more efficiently, but this task is non-trivial since it may require additional static or dynamic analysis features by pointer analysis or dynamic testing [21, 30, 40]. As a future direction, we will investigate to extract detailed code features and incorporate them into our *PerfDectector* for achieving fine-grained bug detection.

Threats to External Validity. One of the threats to external validity is the *generality* of our approach. Our *PerfDectector* mainly focuses on VR/AR projects developed by the Unity framework and mostly implemented in the C# language due to its popularity (§ 2). However, there are some other VR/AR development frameworks, such as Unreal Engine (UE) [31], CryEngine (CE) [44], Blender [44], and so on, among which UE and CE are developed by C++ while Blender is developed by Python. More types of VR/AR projects based on these development frameworks will be considered in the future. Another threat to external validity lies in the selection of eight performance bugs in our approach despite diverse performance bugs existing in VR/AR projects. As shown in § 4.2, 2,430 sample features cannot be categorized into eight defined performance bugs due to larger distances than the threshold, implying that *PerfDectector* is restricted in detecting these well-defined bugs. It is necessary to consider additional types of performance bugs in our *PerfDectector*.

Threats to Internal Validity. During the data labeling phase, we adopt BERT to embed commit messages for semi-supervised learning while the BERT-based embedding may introduce additional noises, thereby degrading the effect of data labeling. To address this issue, we have filtered out noises by removing the samples containing more than two buggy functions. We have verified the

```
public override void OnGUI()
{
    base.OnGUI();
    ...
    if (GUILayout.Button("Create New Level")) CreateSampleLevel();
    GUILayout.Space(10);
    GUILayout.Label("The following actions should be performed once after an update of the SDK was installed,
      ↪ since the framework constantly evolves.", EditorStyles.wordWrappedLabel);
    if (GUILayout.Button("Setup Tags")) SetupTags();
    if (GUILayout.Button("Update/Restore Tiled Data")) RestoreTiled();
}
```

Fig. 15. Case 2: Incorrect detection of ORU bug

accuracy of data labeling by randomly sampling partial data and the accuracy is acceptable (97.45%). Despite this countermeasure, there may exist a minor threat to internal validity.

6 RELATED WORK

Performance Issues in VR/AR Projects. Performance issues in VR/AR projects have become a hot research topic with the proliferation of VR/AR apps [16]. Several empirical studies have been conducted on open-source VR/AR projects to investigate performance-related issues. For example, Nusrat et al. [27] proposed an empirical study about performance optimization in Unity-based VR projects, in which the performance optimization issues were divided into 11 categories and their impact on the life cycle of VR projects was also investigated. Hogan et al. [12] proposed an empirical study about performance optimization of UE-based Extended Reality (XR) projects and identified 14 performance bugs in their study. They also proposed a static analysis-based detection method by pattern matching for performance-bug detection. Rzig et al. [34] analyzed test case distribution in VR open-source projects, in which some test cases are related to performance detection. Li et al. [17] also proposed an exploratory study of WebXR bugs and identified some performance-related bugs. By contrast, our work focuses on detecting performance bugs in Unity-based VR/AR projects.

AI-based Bug Detection. Bug detection or bug prediction has been a critical topic in the software engineering community [36]. Both static analysis [2, 11] and dynamic testing [30] approaches have been proposed to detect bugs. Recently, AI-based approaches have become one of the mainstream approaches in bug detection with the rapid development of DL and LLM. For DL, Chen et al. [4] proposed a CNN-based approach to detect UI glitch bugs in graphically rich applications, such as VR games. Tang et al. [37] proposed a function-level vulnerability detection approach by adopting GNN to embed code statements as well as control flow graphs (CFG) and multilayer perceptron to achieve SODA binary classification in the public dataset CodeXGLUE [22]. As for LLM, Gomes et al. [9] proposed a comparative study between BERT and TF-IDF. They found that the BERT-based approach performs better than TF-IDF in capturing code semantics and predicting long-lived bugs. In our research, we conduct an LLM-based approach for performance bug detection.

7 CONCLUSION

Diverse VR/AR apps have been increasingly deployed across VR/AR devices to provide users with an immersive experience. However, little attention has been paid to performance bugs, which may profoundly affect users' experience. In this paper, we define eight types of VR/AR performance bugs based on 1,013 projects collected from open-source repositories. We then adopt semi-supervised learning to label the collected data. We next propose an LLM-based approach, which can learn AST, object invocation, and source code from the constructed performance-bug dataset. Extensive experiments demonstrate the superior performance of the proposed method to conventional and LLM-based methods. The constructed dataset and proposed LLM-based method will pave the ground to guarantee the software quality of VR/AR apps.

REFERENCES

[1] Tamás Aladics, Judit Jász, and Rudolf Ferenc. 2021. Bug Prediction Using Source Code Embedding Based on Doc2Vec. In *Computational Science and Its Applications – ICCSA 2021*, Osvaldo Gervasi, Beniamino Murgante, Sanjay Misra, Chiara Garau, Ivan Blečić, David Taniar, Bernady O. Apduhan, Ana Maria A. C. Rocha, Eufemia Tarantino, and Carmelo Maria Torre (Eds.). Springer International Publishing, Cham, 382–397.

[2] Nathaniel Ayewah, William Pugh, David Hovemeyer, J. David Morgenthaler, and John Penix. 2008. Using Static Analysis to Find Bugs. *IEEE Software* 25, 5 (2008), 22–29. https://doi.org/10.1109/MS.2008.130

[3] Polona Caserman, Augusto Garcia-Agundez, and Stefan Göbel. 2020. A Survey of Full-Body Motion Reconstruction in Immersive Virtual Reality Applications. *IEEE Transactions on Visualization and Computer Graphics* 26, 10 (2020), 3089–3108. https://doi.org/10.1109/TVCG.2019.2912607

[4] Ke Chen, Yufei Li, Yingfeng Chen, Changjie Fan, Zhipeng Hu, and Wei Yang. 2021. GLIB: Towards Automated Test Oracle for Graphically-Rich Applications. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1093–1104. https://doi.org/10.1145/3468264.3468586

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. 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, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[6] Paul E. Dickson, Jeremy E. Block, Gina N. Echevarria, and Kristina C. Keenan. 2017. An Experience-Based Comparison of Unity and Unreal for a Stand-Alone 3D Game Development Course. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education* (Bologna, Italy) *(ITiCSE '17)*. Association for Computing Machinery, New York, NY, USA, 70–75. https://doi.org/10.1145/3059009.3059013

[7] Georgia Drampalou, Nikolaos Kourniatis, and Ioannis Voyiatzis. 2023. Customized Toolbox in VR Design. In *Proceedings of the 26th Pan-Hellenic Conference on Informatics* (Athens, Greece) *(PCI '22)*. Association for Computing Machinery, New York, NY, USA, 14–20. https://doi.org/10.1145/3575879.3575960

[8] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, Online, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[9] Luiz Gomes, Ricardo da Silva Torres, and Mario Lúcio Côrtes. 2023. BERT- and TF-IDF-based feature extraction for long-lived bug prediction in FLOSS: A comparative study. *Information and Software Technology* 160 (2023), 107217. https://doi.org/10.1016/j.infsof.2023.107217

[10] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie LIU, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *International Conference on Learning Representations*. https://openreview.net/forum?id=jLoC4ez43PZ

[11] Andrew Habib and Michael Pradel. 2018. How many of all bugs do we find? a study of static bug detectors. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE '18)*. Association for Computing Machinery, New York, NY, USA, 317–328. https://doi.org/10.1145/3238147.3238213

[12] Jason Hogan, Aaron Salo, Dhia Elhaq Rzig, Foyzul Hassan, and Bruce Maxim. 2022. Analyzing Performance Issues of Virtual Reality Applications. *arXiv preprint arXiv:2211.02013* (2022).

[13] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2023. Large Language Models for Software Engineering: A Systematic Literature Review. *arXiv preprint arXiv:2308.10620* (2023).

[14] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep Code Comment Generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)* (Gothenburg, Sweden). ACM, New York, NY, USA, 200–210. https://doi.org/10.1145/3196321.3196334

[15] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1412.6980

[16] Shuqing Li, Lili Wei, Yepang Liu, Cuiyun Gao, Shing-Chi Cheung, and Michael R Lyu. 2023. Towards Modeling Software Quality of Virtual Reality Applications from Users' Perspectives. *arXiv preprint arXiv:2308.06783* (2023).

[17] Shuqing Li, Yechang Wu, Yi Liu, Dinghua Wang, Ming Wen, Yida Tao, Yulei Sui, and Yepang Liu. 2020. An Exploratory Study of Bugs in Extended Reality Applications on the Web. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. 172–183. https://doi.org/10.1109/ISSRE5003.2020.00025

[18] Yong Li and Wei Gao. 2019. DeltaVR: Achieving High-Performance Mobile VR Dynamics through Pixel Reuse. In *Proceedings of the 18th International Conference on Information Processing in Sensor Networks* (Montreal, Quebec, Canada) *(IPSN '19)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/3302506.3310385

[19] Yen-Chun Li, Chia-Hsin Hsu, Yu-Chun Lin, and Cheng-Hsin Hsu. 2020. Performance Measurements on a Cloud VR Gaming Platform. In *Proceedings of the 1st Workshop on Quality of Experience (QoE) in Visual Multimedia Applications* (Seattle, WA, USA) *(QoEVMA'20)*. Association for Computing Machinery, New York, NY, USA, 37–45. https://doi.org/10.1145/3423328.3423497

[20] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2020. RoBERTa: A Robustly Optimized BERT Pretraining Approach. https://openreview.net/forum?id=SyxS0T4tvS

[21] V. Benjamin Livshits and Monica S. Lam. 2003. Tracking pointers with path and context sensitivity for bug detection in C programs *(ESEC/FSE-11)*. Association for Computing Machinery, New York, NY, USA, 317–326. https://doi.org/10.1145/940071.940114

[22] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, MING GONG, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie LIU. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*. https://openreview.net/forum?id=6lE4dQXaUcb

[23] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *International Conference on Learning Representations*. https://openreview.net/forum?id=idpCdOWtqXd60

[24] Subrota Kumar Mondal, Yu Pei, Hong Ning Dai, H M Dipu Kabir, and Jyoti Prakash Sahoo. 2020. Boosting UI Rendering in Android Applications. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. 285–286. https://doi.org/10.1109/QRS-C51114.2020.00055

[25] David L Neumann, Robyn L Moffitt, Patrick R Thomas, Kylie Loveday, David P Watling, Chantal L Lombard, Simona Antonova, and Michael A Tremeer. 2018. A systematic review of the application of interactive virtual reality to sport. *Virtual Reality* 22 (2018), 183–198.

[26] Paweł Nowacki and Marek Woda. 2020. Capabilities of ARCore and ARKit Platforms for AR/VR Applications. In *Engineering in Dependability of Computer Systems and Networks*, Wojciech Zamojski, Jacek Mazurkiewicz, Jarosław Sugier, Tomasz Walkowiak, and Janusz Kacprzyk (Eds.). Springer International Publishing, Cham, 358–370.

[27] Fariha Nusrat, Foyzul Hassan, Hao Zhong, and Xiaoyin Wang. 2021. How Developers Optimize Virtual Reality Applications: A Study of Optimization Commits in Open Source Unity Projects. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 473–485. https://doi.org/10.1109/ICSE43902.2021.00052

[28] Roy Oberhauser and Carsten Lecon. 2017. Gamified Virtual Reality for Program Code Structure Comprehension. *International Journal of Virtual Reality* 17, 2 (2017).

[29] Zainab Oufqir, Abdellatif El Abderrahmani, and Khalid Satori. 2020. ARKit and ARCore in serve to augmented reality. In *2020 International Conference on Intelligent Systems and Computer Vision (ISCV)*. 1–7. https://doi.org/10.1109/ISCV49265.2020.9204243

[30] Michael Pradel and Thomas R. Gross. 2012. Leveraging test generation and specification mining for automated bug detection without false positives. In *2012 34th International Conference on Software Engineering (ICSE)*. 288–298. https://doi.org/10.1109/ICSE.2012.6227185

[31] Weichao Qiu and Alan Yuille. 2016. UnrealCV: Connecting Computer Vision to Unreal Engine. In *Computer Vision – ECCV 2016 Workshops*, Gang Hua and Hervé Jégou (Eds.). Springer International Publishing, Cham, 909–916.

[32] Jaziar Radianti, Tim A. Majchrzak, Jennifer Fromm, and Isabell Wohlgenannt. 2020. A systematic review of immersive virtual reality applications for higher education: Design elements, lessons learned, and research agenda. *Computers & Education* 147 (2020), 103778. https://doi.org/10.1016/j.compedu.2019.103778

[33] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research* 21, 140 (2020), 1–67. http://jmlr.org/papers/v21/20-074.html

[34] Dhia Elhaq Rzig, Nafees Iqbal, Isabella Attisano, Xue Qin, and Foyzul Hassan. 2023. 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* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1269–1281. https://doi.org/10.1145/3597926.3598134

[35] Luciano Sampaio and Alessandro Garcia. 2016. Exploring context-sensitive data flow analysis for early vulnerability detection. *Journal of Systems and Software* 113 (2016), 337–361. https://doi.org/10.1016/j.jss.2015.12.021

[36] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. 2014. Bug characteristics in open source software. *Empirical software engineering* 19 (2014), 1665–1705.

[37] Wei Tang, Mingwei Tang, Minchao Ban, Ziguo Zhao, and Mingjun Feng. 2023. CSGVD: A deep learning approach combining sequence and graph embedding for source code vulnerability detection. *Journal of Systems and Software* 199 (2023), 111623. https://doi.org/10.1016/j.jss.2023.111623

[38] Yang Tang, Hanqing Zhou, and Hang Su. 2022. Automatic Classification of Software Bug Reports Based on LDA and Word2Vec. In *2022 2nd International Conference on Computer Science, Electronic Information Engineering and Intelligent Control Technology (CEI)*. 491–495. https://doi.org/10.1109/CEI57409.2022.9950207

[39] Tevita Tanielu, Raymond 'Akau'ola, Elliot Varoy, and Nasser Giacaman. 2019. Combining Analogies and Virtual Reality for Active and Visual Object-Oriented Programming. In *Proceedings of the ACM Conference on Global Computing Education* (Chengdu,Sichuan, China) *(CompEd '19)*. Association for Computing Machinery, New York, NY, USA, 92–98. https://doi.org/10.1145/3300115.3309513

[40] David A. Tomassi and Cindy Rubio-González. 2021. On the Real-World Effectiveness of Static Bug Detectors at Finding Null Pointer Exceptions. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 292–303. https://doi.org/10.1109/ASE51524.2021.9678535

[41] Evelyn Trainor-Fogleman. 2004. Unity vs Unreal Engine: Game engine comparison guide for 2024. *Evercast* (2004). https://www.evercast.us/blog/unity-vs-unreal-engine

[42] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[43] Sara Vlahovic, Mirko Suznjevic, and Lea Skorin-Kapov. 2022. A survey of challenges and methods for Quality of Experience assessment of interactive VR applications. *Journal on Multimodal User Interfaces* 16, 3 (2022), 257–291.

[44] Chaitya Vohera, Heet Chheda, Dhruveel Chouhan, Ayush Desai, and Vijal Jain. 2021. Game Engine Architecture and Comparative Study of Different Game Engines. In *2021 12th International Conference on Computing Communication and Networking Technologies (ICCCNT)*. 1–6. https://doi.org/10.1109/ICCCNT51525.2021.9579618

[45] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 297–308. https://doi.org/10.1145/2884781.2884804

[46] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 8696–8708. https://doi.org/10.18653/v1/2021.emnlp-main.685

[47] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, Qun Liu and David Schlangen (Eds.). Association for Computational Linguistics, Online, 38–45. https://doi.org/10.18653/v1/2020.emnlp-demos.6

[48] Zixuan Zhang, Feng Wen, Zhongda Sun, Xinge Guo, Tianyiyi He, and Chengkuo Lee. 2022. Artificial Intelligence-Enabled Sensing Technologies in the 5G/Internet of Things Era: From Virtual Reality/Augmented Reality to the Digital Twin. *Advanced Intelligent Systems* 4, 7 (2022), 2100228. https://doi.org/10.1002/aisy.202100228 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/aisy.202100228