# Automated Bug Frame Retrieval from Gameplay Videos Using Multimodal Large Language Models

Wentao Lu
University of Alberta
Edmonton, Canada
wlu4@ualberta.ca

Alexander Senchenko
Electronic Arts
Vancouver, Canada
asenchenko@ea.com

Abram Hindle
University of Alberta
Edmonton, Canada
abram.hindle@ualberta.ca

Cor-Paul Bezemer
University of Alberta
Edmonton, Canada
bezemer@ualberta.ca

## Abstract

Modern game studios deliver new builds and patches at a rapid pace, generating thousands of bug reports, many of which embed gameplay videos. To verify and triage these bug reports, developers must watch the submitted videos. This manual review is labour-intensive, slow, and hard to scale. In this paper, we introduce an automated pipeline that reduces each video to a single frame that best matches the reported bug description, giving developers instant visual evidence that pinpoints the bug.

Our pipeline begins with *FFmpeg* for keyframe extraction, reducing each video to a median of just 1.90% of its original frames while still capturing bug moments in 98.79% of cases. These keyframes are then evaluated by a multimodal large language model (GPT-4o) with visual and text capabilities, which ranks them based on how well they match the textual bug description and selects the most representative frame. We evaluated this approach using real-world developer-submitted gameplay videos and JIRA bug reports from a popular First-Person Shooter (FPS) game. The pipeline achieves an overall F1 score of 0.79 and Success of 0.89 for the top-1 retrieved frame. Performance is highest for the Lighting & Shadow (F1 = 0.94), Physics & Collision (0.86), and UI & HUD (0.83) bug categories, and lowest for Animation & VFX (0.51).

By replacing video viewing with an immediately informative image, our pipeline supports the visual confirmation step of the bug triaging process, offering practical benefits to quality assurance (QA) teams and developers across the game industry.

## CCS Concepts

• **Software and its engineering → Maintaining software**.

## Keywords

issue reports, bug frame retrieval, gameplay video analysis, multimodal large language models

**ACM Reference Format:**
Wentao Lu, Alexander Senchenko, Abram Hindle, and Cor-Paul Bezemer. 2026. Automated Bug Frame Retrieval from Gameplay Videos Using Multimodal Large Language Models. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE-SEIP '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3786583.3786865

## 1 Introduction

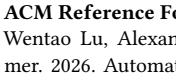
The rapid evolution of digital gaming has led to video games becoming increasingly complex software systems. As developers and testers engage with these systems, they encounter and report thousands of bugs. Among these, visual bugs can be particularly impactful, as they can significantly degrade the player experience. Prior work in traditional software development has shown that reviewing large volumes of bug reports is tedious and time-consuming [5, 9, 17, 42, 52]. For games, this challenge is further amplified by the inherently visual nature of bugs. Reports often include gameplay videos to convey issues that are difficult to describe in text alone. For example, visual bugs such as missing textures, lighting glitches, or character clipping are better presented in a visual manner.

Prior studies have shown that augmenting bug reports with screenshots [6, 47, 50], animated GIFs [12], or videos [9, 33] can improve reproducibility and reduce fix times. The added visual context enables developers to more quickly understand, confirm, and prioritize issues. However, this benefit comes at the cost of increased review time: videos in bug reports often do not pinpoint the bug moment directly, instead including surrounding gameplay context. As a result, bug moments can be difficult to locate, especially when they occur for only a short moment.

In our industrial setting, quality assurance (QA) reviewers already know that a bug exists once a report and video have been submitted. However, they often either (i) manually review the entire video to visually confirm the issue, or (ii) skip this visual confirmation step. Our goal in this work is therefore not to replace the full bug triaging process, but to make this quality-assessment step more efficient by surfacing a representative bug frame.

Recent advances in computer vision and natural language processing have enabled new methods for automated video understanding. In particular, multimodal large language models (MLLMs), such as GPT-4o, Claude, Gemini, and LLaVA, can jointly interpret visual inputs and textual prompts [2, 14, 23, 36]. These models show strong potential to automate tasks traditionally performed by humans, including identifying visual anomalies and verifying gameplay bugs [43, 45, 46].

In this study, we present a novel pipeline (shown in Figure 1) that leverages an MLLM to assist with the bug confirmation step of bug report triage. Our approach reduces each bug report video




*ICSE-SEIP '26, Rio de Janeiro, Brazil*

ACM ISBN 979-8-4007-2426-8/26/04
https://doi.org/10.1145/3786583.3786865

**Bug frames retrieved in gameplay video by GPT-4o**


**Prompt:** Find the frame that matches the bug description.

**Bug description:** Visual bug when Wraith finisher gets interrupted by the portal animation.

**GPT-4o:** Based on the provided frames and prior explanations, frames 000031.jpg, 000032.jpg, and 000065.jpg clearly show the gameplay scenario where the portal animation appears clipped and blocks the player's view, and the visual evidence of the bug is most likely observable. These frames are ranked in descending order of clarity and relevance to the bug description.

**Figure 1: Examples of bug frames identified and retrieved in gameplay video sourced from reddit [40]. The prompt is for demonstration only; the actual prompt design is discussed in a later section.**

to a single, automatically selected frame that best represents the described bug, enabling triagers to quickly confirm a reported bug. Our approach leverages:

- **Keyframes** (snapshots in gameplay videos that capture major visual changes) are used as a down-sampling method to extract important frames from videos. These keyframes are inserted by video encoders and often coincide with major scene transitions, either based on abrupt visual changes (e.g., rendering glitches, camera shifts) or fixed intervals [1, 20, 22]. This makes them a natural choice for gameplay analysis, allowing us to reduce the video to only 1.90% of its frames while preserving most visually distinct moments.
- **An Multimodal LLM** (GPT-4o [36]) ranks the extracted keyframes against the bug description and returns the single frame most likely to depict the reported problem.

We focus on the following research questions:

**RQ1: How accurately can keyframes represent bug moments in gameplay videos?** To ensure that keyframes can indeed capture bug moments, we evaluated keyframe-based extraction using *FFmpeg* [1]. We found that this approach retained at least one bug frame in 98.79% of videos while sampling only 1.90% of all frames (with approximately 10 keyframes per video and an average of 4.79 keyframes that represent the bug). Thus, keyframe extraction offers an efficient down-sampling strategy that still preserves the most representative frames of a visual bug.

**RQ2: How accurately can an MLLM retrieve bug frames from a set of keyframes given a bug description?** A retrieval system must reliably extract the relevant frame that captures the bug from this potentially noisy and varied input. We evaluated our pipeline on 350 videos and achieved an overall 89% Success@1 and F1@1 score of 0.79 for the top returned frame. The pipeline performs best on Lighting & Shadow (F1 = 0.94), Physics & Collision (0.86), and UI & HUD (0.83). This shows that the MLLM can usually retrieve a representative bug frame and thus greatly reduce the effort developers spend reviewing video footage.

**RQ3: How consistent is bug frame retrieval by an MLLM across repeated runs?** Because of the inherent non-determinism of MLLM, it is important to evaluate the consistency of bug frame retrieval across repeated runs. Although the overall F1 score remained stable across runs (median = 0.79), variation in true and false positives highlights the non-deterministic nature of MLLM outputs. For 183 out of 350 videos, the retrieval outcome was consistently correct across all runs. This number increases to 215 when applying a simple majority vote over 3 runs.

The main contributions of our paper are as follows.

- We developed an automated pipeline that integrates MLLM capabilities (GPT-4o) with video frame analysis to return a single frame that matches a textual bug description.
- We show that keyframes from gameplay videos preserve bug-relevant content and can be used as a source of bug frame retrieval.
- We quantify the pipeline's overall success, category-specific performance, and run-to-run stability on real bug reports from an AAA game, providing a comprehensive assessment of its performance and reliability in bug frame retrieval.

**Paper organization:** Section 2 reviews related work. Section 3 introduces our pipeline, including keyframe extraction and MLLM usage, broken down into four steps. Section 4 details the experimental setup. Section 5 answers the research questions. Section 6 compares our results with prior work and discusses the limitations of our approach. Section 7 discusses threats to validity, and Section 8 concludes our work.

## 2 Related Work

Driven by advancements in both data collection (automated gameplay and streamers' footage) [3, 7, 38] and analysis techniques, including recent progress in bug detection and retrieval [43, 45, 46, 48],

gameplay videos are increasingly important in game quality assurance. In this section, we review work in three primary areas: (1) traditional quality assurance in games, (2) bug detection from gameplay videos, and (3) MLLM assistants for video game QA.

## 2.1 Traditional quality assurance in games

Game QA traditionally relies heavily on manual efforts, including playtesting, visual inspection, and reproduction of reported issues by dedicated QA personnel [9, 17, 42]. While effective for identifying critical failures such as game crashes, these workflows are highly labor-intensive and time-consuming [5, 9, 42, 53]. In particular, bug reports often include video recordings as supporting evidence, each of which requires manual review and interpretation. Furthermore, the vast number of possible game states and the non-determinism introduced by factors such as multithreading and randomness make exhaustive manual QA impractical for modern games [17]. These challenges have motivated growing interest in automation and tooling to assist various aspects of the QA process.

**Our work addresses a specific pain point within this landscape** by automating the retrieval of visually relevant frames from gameplay videos associated with bug reports. Rather than replacing testers, our method aims to support QA workflows by surfacing candidate frames for inspection, reducing the need for exhaustive manual scrubbing of video footage.

## 2.2 Bug detection in gameplay videos

Early studies explored the potential of mining gameplay videos to extract information about bugs and glitches. One of the first works in this space is by Lin et al. [25], who investigated whether gameplay videos that showcase bug videos can be automatically identified using publicly available metadata from online platforms with a precision of 0.80. Building on this idea of mining gameplay content, Guglielmi et al. [17, 18] introduced an approach to automatically extract relevant segments from gameplay videos by partitioning videos into meaningful units based on streamer captions, shifting the timestamp by $k$ seconds before the video cuts, and categorizing the detected anomalies by type and context through Machine Learning (ML). Similarly, TrueLove et al. [48] proposed an ML-based method to classify video segments, also split based on video transcript, as buggy or non-buggy, and analyzed visual features that correlate with bug occurrences. Guglielmi et al. [16] introduced *HASTE*, a tool designed to detect stuttering events in gameplay videos. Azizi et al. [5] framed bug detection as an anomaly detection problem and leveraged Long-Short-Term Memory (LSTM) networks to identify perceptual and behavioral bugs in gameplay videos. By modeling temporal dependencies in video sequences, their framework is able to flag unusual frames without relying solely on rule-based methods.

These methods rely heavily on captions or transcripts for video segmentation. This introduces key limitations: (1) such captions do not exist in developer-submitted gameplay footage, and (2) the timing of transcripts may not align precisely with the occurrence of the visual bug, resulting in incomplete or misaligned segment boundaries. Moreover, some techniques focus only on certain bug types (e.g. stuttering), limiting their applicability to the broader range of visual bugs encountered in game development.

While prior methods aim to detect visual bugs through anomaly classification or transcript-driven segmentation, **our goal is fundamentally different: we focus on retrieving visually relevant frames from gameplay videos based on natural language bug descriptions**. To support this, we introduce a deterministic keyframe segmentation strategy that enables frame-level analysis without requiring captions or transcripts. Unlike prior work that often relies on content from streamers or online platforms, our evaluation is conducted on industrial gameplay data, making our findings directly applicable to real-world QA workflows.

## 2.3 MLLM assistants for video game QA

Beyond bug detection, there is growing interest in employing large multimodal models as vision assistants in the video game domain [13, 19, 28]. Taesiri et al. [46] show promising results by leveraging zero-shot transfer capabilities of the Contrastive Language Image Pre-Training (CLIP) model to query bug images. They also introduced VideoGameBunny [43], a model fine-tuned on extensive video game image datasets and associated instruction pairs, demonstrating an improved understanding of the game context and more accurate in-game responses. In subsequent work, Taesiri et al. [44, 45] proposed several benchmark suites, including *VideoGameQA-Bench* and *GlitchBench*, to evaluate the capabilities of state-of-the-art MLLMs across a range of video game QA tasks. These benchmarks assess model performance on tasks such as glitch detection in both static images and gameplay videos, highlighting the potential and current limitations of using large multimodal models for bug understanding/detection in games.

Melhart et al. [29] investigate whether large language models can capture continuous player engagement from gameplay videos. Their comprehensive evaluation examines state-of-the-art models, including variants of GPT-4o [36] and LLaVA [23], using one-shot and few-shot multimodal prompting strategies. Although the study reports that the overall accuracy of LLM-based engagement prediction only marginally surpasses a majority class baseline, it highlights that model performance is highly sensitive to input modality, prompting strategy, and even the specific context of the game. These insights not only underscore the promise of integrating LLMs into game content analysis but also reveal current limitations that motivate further research in autonomous gameplay video interpretation.

While prior work has applied large language models to QA tasks such as glitch classification or engagement prediction, **our method explores a different application: using MLLM for frame-level retrieval guided by natural language bug descriptions.** Specifically, we use GPT-4o in a zero-shot setting [31, 34] to rank keyframes based on their semantic alignment with the bug report description. This decision aligns with constraints in our industrial setting, where QA teams prefer methods that eliminate the need for labeled training data or supplementary example images, thereby limiting token usage and operational costs. Additionally, we evaluate the stability of this retrieval process via multi-run consistency checks, a dimension not explored in previous studies.

# 3 Methodology

This section details our approach for bug frame retrieval in gameplay videos using MLLM. The integrated pipeline consists of four

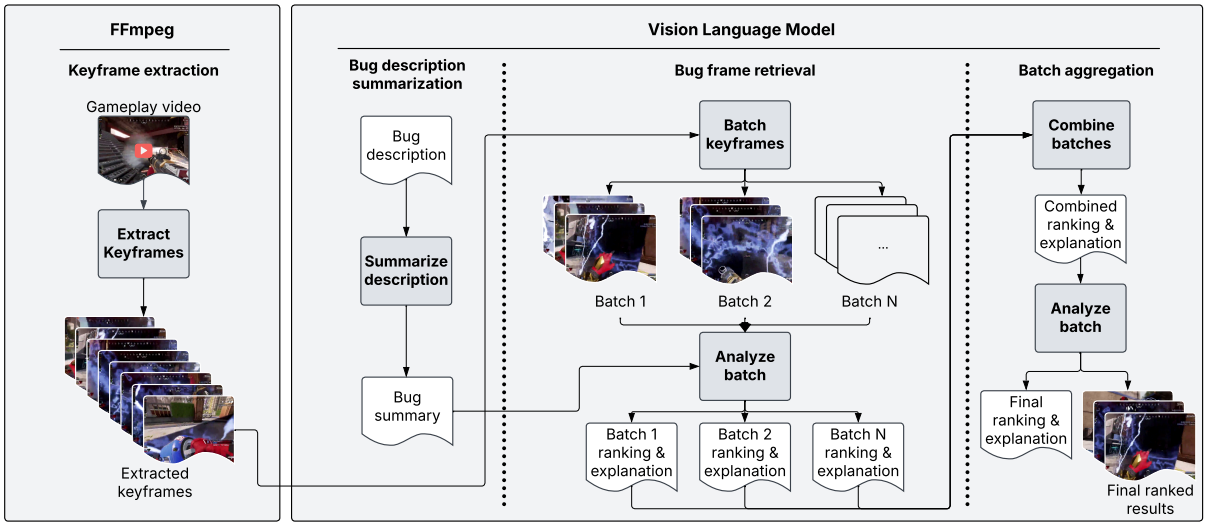



**Figure 2: Overview diagram of our methodology (example video sourced from Reddit [40] as an illustration of the workflow).**

parts: keyframe extraction, bug description summarization, bug frame retrieval, and batch aggregation. We provide an overview of our methodology in Figure 2.

## 3.1 Keyframe extraction

To reduce computational overhead while preserving bug moments in gameplay videos, we adopt a keyframe-based down-sampling strategy. We leverage the internal encoding structure of video files to extract visually significant moments using keyframes.

We utilize *FFmpeg* [1], a widely adopted tool for multimedia processing, to extract keyframes. In general, the original videos uploaded by developers may be encoded using unknown settings (a variety of resolutions and bitrates, such as 1.7Mbps or 38Mbps), and keyframes are determined during video encoding based on various factors [20, 22, 41], including:

- **Scene change detection:** A sharp shift in visual content (e.g., from one scene to another) often triggers a new keyframe.
- **Fixed interval configuration:** Encoders may insert keyframes at fixed time intervals or frame counts (e.g., every 120 frames) in cases where no significant visual change occurs.

As a result, keyframe frequency and distribution vary significantly across videos due to differences in their original encoding settings. Some videos may contain very few keyframes if encoded with long fixed intervals or without scene change detection, which can lead to uneven sampling and the potential omission of bug moments. Therefore, we re-encode videos using *FFmpeg* to enhance bug moment coverage through more informative keyframes. Specifically, we normalize all videos using the H.264 encoder under the medium preset. This normalization ensures that keyframes are inserted consistently across videos.

The output of this step is a set of keyframes for each gameplay video. The number of frames extracted varies, from a few to several hundred (see Section 5.1), depending on the duration and how frequently the visual content of the video changes. These frames represent a compressed visual summary of the video, from which at least one is expected to represent the bug described in the corresponding report. These keyframes are then passed to the GPT-4o model for visual interpretation in the following stages.

## 3.2 Bug description summarization

Before starting visual analysis, we first generate a concise bug description. While JIRA reports include a dedicated description field, in our data, developers often paste large blocks of text into it, including logs, internal tags, build information, and in-game coordinates. These details, while useful for debugging and reproducing a bug, are irrelevant to the task of visual bug frame retrieval and consume valuable input tokens when passed to the MLLM.

To remove excess info in the description and reduce token usage, we use GPT-4o to generate summaries that are concise and short for downstream visual analysis tasks. Each generated summary consists of two parts: (1) a concise restatement of the core issue described in the report, and (2) a binary indication of whether the bug is expected to produce a visible effect in the gameplay video. The second component is critical as it allows the following stages in the pipeline to bypass further visual processing for bugs that are likely of a non-visual nature.

To evaluate the reliability of this binary indication, one of the authors verified whether the indication correctly reflected the nature of the bug in a random sample of 100 GPT-4o-generated summaries. The model's answers showed 97% agreement with human judgment.

## 3.3 Bug frame retrieval

For each gameplay video, the extracted keyframes are passed to GPT-4o alongside the corresponding bug description summary. The model is instructed to perform the following tasks:

- Return none if the bug described in the text is not visually observable in any of the frames.
- Identify frames that visually depict the bug based on the provided summary.

- Rank the identified frames in descending order of how clearly they represent the described bug.

The model is also asked to provide an explanation for its ranking, including visual reasoning grounded in the bug description. The response is returned in a structured JSON format with two keys:

- *Explanation:* A natural language summary justifying the model's frame selection and ranking;
- *Ranking:* A ranked list of frame filenames.

Due to GPT-4o's input token limit, a maximum of 50 images can be processed in a single chat completion [37]. While most gameplay videos in our dataset yield fewer than 50 keyframes, a few outliers (in our studied data, only 2 videos) exceed this threshold. To accommodate such cases, we partition the extracted keyframes into consecutive batches, preserving their original order of sequences. This sequential grouping retains contextual continuity; for instance, if a bug occurs immediately after a player action, the relevant batch is likely to include both the trigger and its visual consequence. This batching process is applied iteratively across all extracted frames in the video. Each batch is processed independently using the same prompt. For these 2 outlier videos, we observed that batch size had a negligible impact on retrieval performance.

### 3.4 Batch aggregation

After processing all batches, we obtain multiple sets of ranked keyframes and explanations for each batch as shown in Figure 2. To combine these partial results and produce a final, globally consistent ranking of bug-relevant keyframes, we aggregate all candidate keyframes from the batch-level rankings and their rationales into a single prompt and pass it to GPT-4o again, following recent advances in which large language models can be used as judges [10, 15, 24, 54]. The model is instructed to re-rank all keyframes selected in each batch based on how clearly they depict the bug described in the original JIRA description and by referencing each of the batch-level explanations.

The model's response is returned in the same structured JSON format as above. This step applies even if there is only one batch.

## 4 Experimental setup

In this section, we first describe our experimental setup and then outline the performance metrics used in the evaluation.

**Used models:** We used OpenAI's GPT-4o [36] and GPT-4o-mini [35], accessed through API version "2024-10-21" (at the time stable GA version) [30], as well as the open source Mistral-7B model [32]. GPT-4o was selected for its strong performance in vision-language tasks, particularly for detecting visual glitches in game images [45]. GPT-4o-mini and Mistral-7B are used in the clustering stage of our study as discussed below.

**Data collection:** We utilize real-world industrial game development data logged in JIRA, a widely used platform for bug tracking, issue tracking, and agile project management [4]. Specifically, our data consists of internal JIRA bug reports from a popular First-Person Shooter (FPS) game. Each report typically contains a textual bug description and one or more short gameplay videos (usually under one minute), which are submitted by developers or QA testers to demonstrate the bug. For legal reasons, we are unable to publish our data.

To evaluate retrieval success and category-based performance in a consistent manner, we constructed a subset of 350 gameplay videos as follows:

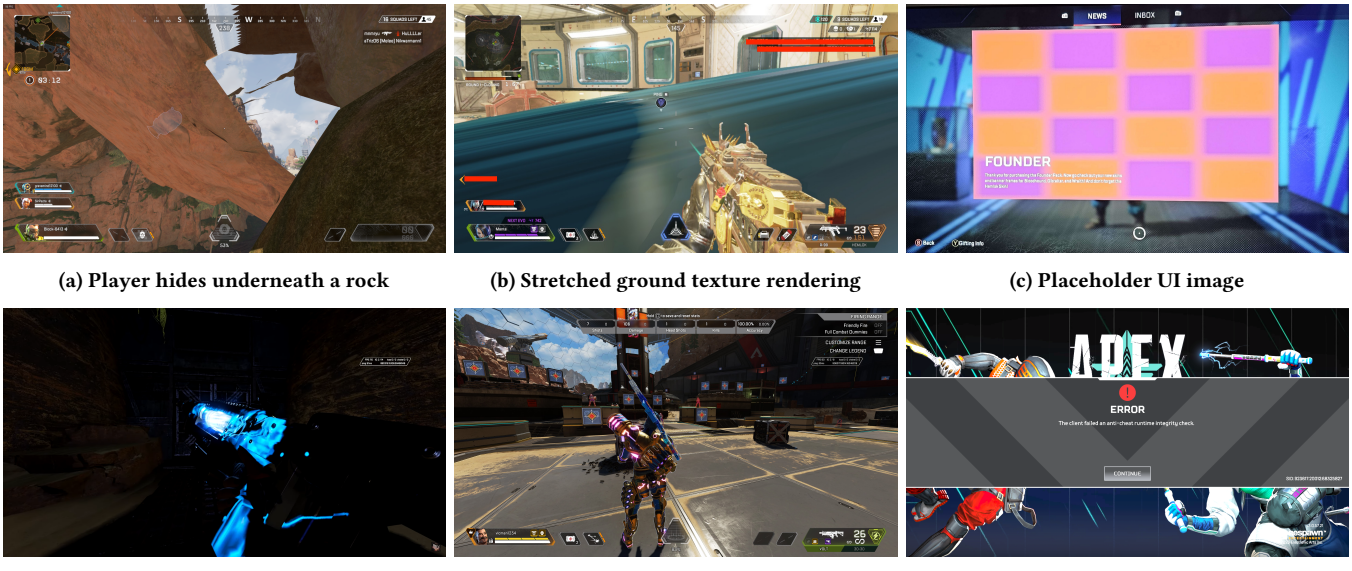
- We first summarized all JIRA bug reports from the past two years into preliminary keyword-based clusters using the Mistral-7B model. This initial step with an open source model was designed to reduce the computational cost of the GPT model due to the large number of reports.
- While Mistral-7B was effective for initial keyword-based clustering, it lacked the summarization quality needed to produce concise category names. We therefore used GPT-4o-mini to refine these clusters into seven high-level bug categories. The final category names are shown in Table 1.
- We then sampled 2,880 JIRA bug reports from an internal dataset collected between June and July 2025. Each sampled report contained at least one gameplay video attachment and was automatically assigned to one of the seven categories based on its textual description using GPT-4o-mini.
- From each of the seven categories, we sampled 50 video samples. One of the authors manually verified each sample to ensure it clearly aligned with the assigned category.
- We manually annotated the keyframes in the videos as bug or non-bug, if any. For each bug report and its associated video(s), the first author watched the full video and read the bug description. Then, the author labeled each keyframe as bug or non-bug depending on whether it visually shows the bug described in the report. This is a non-subjective task in our setting, as the existence and nature of the bug are already clearly stated. Note that several keyframes may represent the bug accurately. These keyframes will serve as ground truth frames in later evaluation.

These 350 videos are used in RQ2 and RQ3. For RQ1, a subset of 247 (manually confirmed to contain a clear visual bug) out of the 350 videos is used to evaluate keyframe extraction effectiveness, as discussed in Section 5.1.

**Performance evaluation:** To evaluate the effectiveness of our bug frame retrieval pipeline, we consider two primary use cases: **Top-1 retrieval** and **Top-N retrieval**. These correspond to different levels of automation in QA workflows.

- In the **Top-1 retrieval** use case, only the highest-ranked frame returned by the LLM is evaluated. This use case is aligned with scenarios where a fully automated system is desired, for example, automatically generating bug reports without human intervention. High precision in this setting is essential, as the top result alone must accurately represent the described bug.
- In the **Top-N retrieval** use case, we consider the first N-ranked frames. This reflects a use case in game development pipelines, where a human is still in the loop. For example, a QA engineer might inspect just the top 3 ranked keyframes to verify bugs mentioned in the report instead of watching a full gameplay video. In such workflows, the key objective is to ensure that at least one of the returned keyframes displays the described bug, reducing the time required to manually review entire videos and triage bug reports.

To assess the correctness of results returned by the MLLM, we compute the following metrics:




(a) Player hides underneath a rock (b) Stretched ground texture rendering (c) Placeholder UI image

(d) Scene too dark (e) Weapon floats over the character's back (f) Game crash error

**Figure 3: Six samples of visual bugs in game videos: (a) Physics & Collision, (b) Rendering & Texture, (c) Camera & UI, (d) Lighting & Shadow, (e) Animation & VFX, (f) Game Crash & Logic. All images are captured from online sources [11].**

**Table 1: Bug Category Definitions and Sample Counts**

| Category | Description | #Samples |
|---|---|---|
| Physics & Collision | Issues arising from the physics engine or collision detection (e.g. objects/players passing through geometry). | 671 |
| Animation & VFX | Problems with character or object animations and visual effects (e.g. body distortion, particle glitches). | 861 |
| Rendering & Texture | Errors in the rendering pipeline or asset texturing (e.g. missing textures, incorrect material artifacts). | 418 |
| UI & HUD | Bugs in Heads-Up Display and User-Interface elements (e.g. HUD misalignment and UI images). | 374 |
| Lighting & Shadow | Faults in scene lighting or post-process effects (e.g. incorrect shadows, abnormally dark environment). | 66 |
| Game Crash & Logic | Game crashes, incorrect game logic flows or functionality behavior (e.g. error messages, player rewards tracking). | 295 |
| Performance | Performance degradations (e.g. Frame-rate drops, stuttering). | 137 |
| Unspecified | These reports were automatically generated by scripts and did not have categories specified. | 58 |

**Table 2: Bug frame retrieval performance by category**

| | #Bug | TP (Top N) | | | Retrieval Outcomes | | | Invalid | Success@N | | | F1 Score |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Category | Videos | 1 | 2 | 3 | TN | FP | FN | (FP + TN) | @1 | @2 | @3 | @1 |
| Physics & Collision | 36 | 28 | 31 | 32 | 9 | 7 | 2 | 8 (5 + 3) | 0.93 | 0.94 | 0.94 | 0.86 |
| Animation & VFX | 23 | 11 | 16 | 16 | 13 | 19 | 2 | 21 (14 + 7) | 0.85 | 0.89 | 0.89 | 0.51 |
| Rendering & Texture | 32 | 22 | 27 | 29 | 8 | 13 | 0 | 16 (10 + 6) | 1.00 | 1.00 | 1.00 | 0.77 |
| UI & HUD | 39 | 31 | 32 | 32 | 5 | 11 | 2 | 6 (6 + 0) | 0.94 | 0.94 | 0.94 | 0.83 |
| Performance | 38 | 20 | 20 | 20 | 11 | 6 | 13 | 2 (1 + 1) | 0.61 | 0.61 | 0.61 | 0.68 |
| Lighting & Shadow | 49 | 40 | 44 | 44 | 1 | 5 | 0 | – | 1.00 | 1.00 | 1.00 | 0.94 |
| Game Crash & Logic | 30 | 17 | 20 | 22 | 17 | 9 | 2 | 4 (3 + 1) | 0.89 | 0.91 | 0.92 | 0.76 |
| **Overall** | **247** | **169** | **190** | **195** | **64** | **70** | **21** | **57 (39 + 18)** | **0.89** | **0.90** | **0.90** | **0.79** |

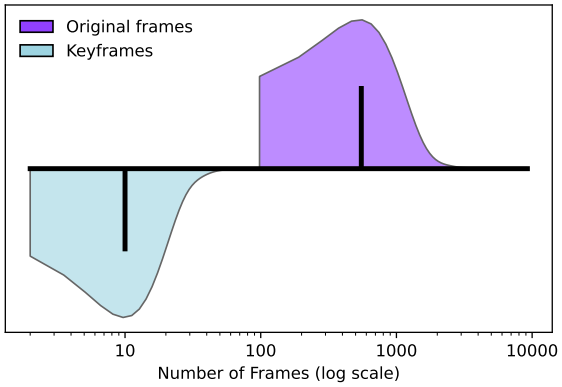



**Figure 4: Total number of frames vs keyframes per video across 247 videos in log scale (bar indicates median value)**

- **Success@N:** The fraction of videos for which a ground truth bug frame appears among the top $N$ retrieved frames. Formally,
$$\text{Success@}N = \frac{\text{TP@}N}{\text{TP@}N + \text{FN}}$$
- **F1 Score@1:** Defined at Top-1 retrieval as:
$$\text{F1@1} = 2 \times \frac{\text{TP@1}}{2 \cdot \text{TP@1} + \text{FP} + \text{FN}}$$
where TP@1 denotes true positives at top-1.
*Note:* In our setup, false positives include those arising from invalid cases, and any retrieved frame that does not correspond to a valid bug frame, regardless of its rank. Therefore, FP@1 = FP@2 = FP@3 = FP, as well as FN, are not position-specific.

We defined the components of the confusion matrix as follows:

- **True Positive (TP):** The model successfully retrieved a bug frame.
- **True Negative (TN):** No visual bug is present in the video, and the model correctly returned no results.
- **False Positive (FP):** The model incorrectly identified a non-bug frame as a bug frame.
- **False Negative (FN):** A visual bug is present in the video, but the model failed to return any frame.
- **Invalid:** All keyframes extracted from the video did not include the bug moment. In such cases, the correct model behavior is to return no result (i.e. treated as TN if none were returned and FP otherwise).

We define the following metrics:

- **Bug coverage:** The percentage of videos for which at least one extracted keyframe depicts the ground-truth bug.
- **Successful retrieval rate:** The proportion of bug-containing videos or reports where the pipeline successfully retrieves at least one frame that matches the ground-truth bug. Unlike Success@N, FP is included.

We release our pipeline implementation, including detailed FFmpeg commands, prompts and configurations at [27].

## 5 Results

In this section, we answer our research questions.

### 5.1 RQ 1: How accurately can keyframes represent bug moments in gameplay videos?

*Motivation:* By manually analyzing the effectiveness of keyframe-based video segmentation, we assess whether extracting keyframes serves as an effective down-sampling technique for preserving bug-relevant content.

*Approach:* We first curated a set of 247 of the studied 350 videos that contain visible visual bugs (bugs that can be easily spotted without ambiguity). Each video was manually verified by one of the authors to ensure it included at least one identifiable bug frame. This step guaranteed that the videos actually contained valid bug frames that could be extracted as keyframes. Second, for each video, we manually reviewed all keyframes produced by the extraction stage and recorded three metrics: (1) the number of failures (defined as cases where no keyframe captured the bug), (2) the average keyframe extraction rate (percentage of keyframes among all frames in a video) and (3) the average number of keyframes per video that actually show the visual bug. To assess how video encoding configurations affect coverage, we conducted this analysis on the videos before and after re-encoding (as described in Section 3.1).

*Findings:* **After re-encoding, bug coverage reaches 98.79% with a keyframe extraction rate of 1.90%.** Given a median video length of 551 frames (Figure 4), a keyframe extraction rate of 1.90% corresponds to approximately 10 keyframes per video, *of which an average of 4.57 keyframes actually depict the bug*. Figure 4 illustrates that the number of extracted keyframes is substantially lower than the total frame count per video. To quantify this reduction, we computed the Wilcoxon signed-rank test (a non-parametric test that examines whether the median paired difference $W$ equals zero) and the Cliff's Delta ($\delta$) effect size between the distributions of total frames and extracted keyframes [8, 51]. The two tests yielded $W = 0$ and a large effect size (Cliff's $\delta = 0.99$). These findings demonstrate that keyframe extraction is an effective down-sampling strategy.

**Extracting keyframes from the original videos achieves only 65.99% bug coverage at 1.10% keyframe extraction rate.** When using each video's original format, keyframes could not be extracted at all in 46 out of 247 videos, due to missing internal metadata or incompatible encoder settings. In the remaining 201 videos, 38 failed to include any keyframes that depicted the described bug. Even when excluding the 46 extraction failures, the remaining videos yield only 81.09% coverage. On average, 1.10% of frames were extracted per video (approximately 6 keyframes), and *only 1.93 of those keyframes, on average, showed the bug*. These results suggest that lower bug coverage in the default setting is due to both (a) an insufficient number of keyframes extracted and (b) complete extraction failure in some videos. Original encoders may use long fixed keyframe intervals, low scene change sensitivity, or non-standard encoder formats that limit FFmpeg's ability to extract keyframes. These findings highlight the importance of re-encoding, not just to increase keyframe density but to ensure keyframe extraction succeeds at all.

> *Keyframes can accurately capture bug moments in gameplay videos. Re-encoding developer-submitted videos can significantly improve the keyframe extraction process.*

## 5.2 RQ 2: How accurately can an MLLM retrieve bug frames from a set of keyframes given a bug description?

*Motivation:* Understanding how accurately the model retrieves bug-relevant frames from a set of keyframes provides key insight into the practical utility of our pipeline. Specifically, it reflects how well the model can help locate the frame that matches the given bug description. Meanwhile, by evaluating the ability to retrieve bug frames in different categories of bugs, we can better understand its strengths and limitations and identify which types of visual bugs are most suitable for this type of automated analysis.

*Approach:* In practice, developers often attach multiple videos to a single report. In our full dataset of 2,880 bug reports, approximately **35%** of reports contain more than one attached video. To account for this, we evaluated retrieval success at two levels:

- **Per-video success:** We evaluate the selected set of 350 videos as discussed in Section 4 and assess performance under two retrieval settings: **Top-1** and **Top-N**.
- **Per-report success:** Not every video necessarily contains the reported visual bug, e.g. because developers may upload videos to demonstrate correct game behavior for comparison. To assess performance at the report level, we randomly selected 100 JIRA bug reports containing multiple video attachments and evaluated whether our pipeline could retrieve at least one relevant bug frame from any of the associated videos.

We then conducted **per-category analysis:** Each of the 50 samples from the 7 categories (Table 1) is sent to our pipeline to obtain ranked bug frame results. For each sample, we assess the retrieval outcome against our manual annotation (ground truth keyframes containing visible bugs) and compute both the success and F1 scores to quantify category-specific performance.

Finally, to provide a point of reference, we compare our pipeline against **two naive baselines**: selecting the first extracted keyframe and selecting a random keyframe from the 247 bug videos.

*Findings:* **Per-video retrieval has an F1@1 score of 0.79 and a Success@1 of 0.89.** Table 2 shows that the model successfully retrieved a correct bug frame at the top-1 result for 169 out of 247 bug videos, yielding a Success@1 of 0.89. **For comparison, naive baselines perform worse: selecting only the first keyframe in each video yields 0.26 Success@1, and randomly selecting a keyframe achieves 0.29 (mean over 5 runs).** When considering the top-3 results, performance improves to 195 videos, corresponding to a Success@3 of 0.90. Across the evaluation set, we observe 70 false positives and 21 false negatives, resulting in an overall F1@1 score of 0.79. The successful retrieval rate is 68.42%. The results indicate that the top-ranked frame is most often the correct one. In cases where multiple frames are selected, the matched bug frame always appears within the top 3 results.

**Per-report retrieval improves successful retrieval rate from 68.42% to 82.14%.** We randomly sample 100 multi-video JIRA bug reports from the 2,880 report dataset and include all their attached videos, yielding 337 videos in total. Our pipeline's retrieval over these 337 videos results in 204 TP, 95 FP, 15 FN, and 23 TN. We identified 84 reports containing at least one video with a visible bug. Our pipeline successfully retrieved at least one correct bug frame in 69 of these 84 reports (82.14%). While this does not affect the overall F1@1 score (which remains at 0.79), aggregating retrieval at the report level improves the chance of retrieving at least one valid bug frame per report.

**Lighting & Shadow achieves the highest retrieval Success (F1@1 = 0.94)**, followed by Physics & Collision and UI & HUD. Table 2 shows that the model achieves its highest performance on the Lighting & Shadow category, retrieving bug frames more reliably than in other categories. Physics & Collision and Camera & UI also maintain high true-positive counts with low false-negative counts. Several common factors contribute to this higher performance trend:

- **High visual contrast:** Bugs exhibit dramatic visual changes in the game, such as extremely dark scenes (Figure. 3d), collisions with surrounding objects (Figure. 3a), or placeholder images/text in UI elements that have colors that stand out from the rest of the environment (Figure. 3c).
- **Strong text–visual alignment:** Bug descriptions align well with visible content (e.g., "character can clip underground" or "placeholder text appears in the menu"), making it easier for the model to retrieve a correct frame.

Our analysis reveals several key limitations that hinder retrieval performance:

- **Flickering events:** A major issue among rendering-related bugs is brief flickers of objects (e.g., a light source at a distance starts flickering). While these are easily noticeable during video playback, a single extracted frame often fails to capture the flicker.
- **On-screen debug messages:** While videos in other categories include debug overlays as well, they're not defining features of those categories. These messages particularly affect **Game Crash & Logic** videos. Development logs and messages often dominate the visual focus and may redirect MLLM to producing irrelevant retrieval.
- **Abstract bugs:** Logic-related issues, such as a reward system not updating after an in-game action, often fall under **Game Crash & Logic**. These bugs lack a clear visual signal and usually require reasoning across multiple frames to understand cause and effect.
- **Post-processing effects mistaken as false positives:** Visual effects such as motion blur, bloom, and lens flares are prevalent in **Animation & VFX** videos and usually trigger a new keyframe due to a major shift in visual content. These artistic effects are sometimes misinterpreted by the model as visual glitches, increasing false positive rates when analyzing isolated frames.
- **Minimal visual anomalies:** While subtle artifacts can appear across categories, bugs in Animation & VFX often involve minor animation glitches that are difficult to observe (Figure 3e).
- **Frame-rate drops:** Most **Performance** bugs involve degraded frame rates during gameplay. Although developer-submitted videos often include an on-screen FPS indicator, the expected frame rate is rarely specified, and these drops often do not trigger new keyframes due to minimal visual changes. As a result, the model struggles to identify frame-rate issues, especially in modern games where high frame

rates are standard. For instance, a drop to 90 FPS may indicate poor performance if the game is expected to run at 120 FPS, just as 40 FPS would be problematic in a game targeting 60 FPS. Without explicit context, the model cannot reliably infer whether the observed frame rate reflects a bug.

*Our pipeline achieves a Success@1 of 0.89 and an F1@1 score of 0.79 at the per-video level and retrieves at least one bug frame in 82.14% of multi-video reports. Retrieval is highest in **Lighting & Shadow**, **Physics & Collision**, and **UI & HUD** categories, and the weakest in **Performance** and **Animation & VFX**. These results demonstrate the practical effectiveness of MLLM-based retrieval of bug frames in gameplay videos and the potential to considerably reduce video viewing time.*

**Table 3: Retrieval performance across three identical runs**

| | TP@1 | TP@2 | TP@3 | FP | TN | FN | F1@1 |
|---|---|---|---|---|---|---|---|
| 1 | 169 | 190 | 195 | 70 | 64 | 21 | 0.79 |
| 2 | 180 | 194 | 194 | 72 | 62 | 22 | 0.79 |
| 3 | 179 | 192 | 192 | 76 | 58 | 24 | 0.78 |
| **Total** | 528 | 576 | 581 | 218 | 184 | 67 | |

## 5.3 RQ 3: How consistent is bug frame retrieval by an MLLM across repeated runs?

*Motivation:* A known limitation of large language models is their non-deterministic behavior: repeated queries to the same model can yield different outputs, even when the input remains unchanged. This variability stems from the probabilistic nature of their design and sampling mechanisms [21, 26, 39, 49]. While some degree of output variation is expected, it raises concerns about the reliability of our pipeline. Therefore, we assess how consistent the retrieval results are across multiple runs of the same prompt and setup.

*Approach:* We re-run the same set of 350 videos two additional times, resulting in a total of three runs under identical experimental conditions. For each run, we assess the retrieval results at **Top-1** using the same evaluation metrics as before. This allows us to measure the consistency of the pipeline's outputs across repeated executions. To quantify the likelihood of successful retrieval across multiple runs, we compute **pass@k**, which denotes the probability that a correct bug frame is retrieved in at least one of the k runs.

*Findings:* As shown in Table 3, the F1@1 score varies minimally across the three runs, with a median of 0.79. However, despite this stability in the overall metrics, there is some inconsistency across runs for many of the videos. For example, a bug frame might be extracted correctly in run 1 but incorrectly in run 2.

- **# of videos correct in 3/3 runs:** A total of 183 (140 TP and 43 TN) videos had correct retrieval outcomes in all runs.
- **# of videos correct in 2/3 runs:** 48 (42 TP and 6 TN) videos had correct retrieval outcomes in two of the three runs.
- **# of videos correct in 1/3 runs:** 25 (24 TP and 1 TN) videos had correct retrieval outcomes in only one of the runs.
- **# of videos correct in 0/3 runs:** 94 videos including FP and FN. Since only TP@1 is considered, videos where the correct bug frame was retrieved at rank 2 or 3 are considered as incorrect in this analysis.

**Single-run success (*pass@1*):** The *pass@1* metric captures the probability that the pipeline retrieves a correct result in a single run@1. It is defined as:

$$\text{pass@1} = \frac{\text{TP@1 + TN across 3 runs}}{\text{\#Videos across 3 runs}} = \frac{528 + 184}{350 \times 3} \approx 0.68.$$

**Three-run success (pass@3):** The probability that *at least one* of the three runs@3 succeeds:

$$\text{pass@3} = 1 - (1 - \text{pass@1})^3 \approx 0.98$$

This means that under one run, there is a 68% chance that the top-1 retrieved frame is a correct result. This probability increases to 98% under 3 runs.

**Expected value& variation of correctness:** We define the variance of the random variable $X \in \left\{0, \frac{1}{3}, \frac{2}{3}, 1\right\}$ and the expected number of successful retrievals per video as:

$$\mathbb{E}(X) = \frac{94 \cdot 0 + 25 \cdot \frac{1}{3} + 48 \cdot \frac{2}{3} + 183 \cdot 1.0}{350} \approx 0.63$$

To quantify the variability in correctness:

$$\text{Var}(X) = \mathbb{E}(X^2) - \mathbb{E}(X)^2$$

$$\mathbb{E}(X^2) = 0^2 \cdot \frac{94}{350} + \left(\frac{1}{3}\right)^2 \cdot \frac{25}{350} + \left(\frac{2}{3}\right)^2 \cdot \frac{48}{350} + 1^2 \cdot \frac{183}{350} \approx 0.59$$

$$\text{Var}(X) = 0.59 - (0.63)^2 \approx 0.19$$

$$\text{SD}(X) = \sqrt{\text{Var}(X)} \approx 0.44$$

The expected value $\mathbb{E}(X) \approx 0.63$ means that, on average, each video yields correct retrievals in roughly 1.89 ($0.63 \times 3$) out of 3 runs. The variance $\text{Var}(X) \approx 0.19$ and corresponding standard deviation 0.44 indicate considerable variability in correctness, and that using a single run per video may lead to unreliable results.

**Variance mitigation recommendation:** To mitigate retrieval instability across runs, we recommend executing the pipeline multiple times (e.g., three runs per video) and applying a simple majority vote over the retrieved frames. This strategy helps improve consistency without requiring access to ground truth:

- If the same bug frame or no result appears in at least two of the three runs, treat the result as reliable.
- Otherwise, flag the video as *ambiguous* and consider reviewing it manually.

This approach filters out inconsistent results. While it introduces some computational overhead, it improves practical confidence in retrieval quality. In our evaluation, applying majority voting would raise the number of stable results from 183 to 215 out of 350 videos.

*Retrieval performance remains stable (median F1@1 = 0.79), but per-video results vary.* A majority vote over 3 runs helps improve consistency by confirming stable results and flagging ambiguous ones.

## 6 Discussion

**Comparison with prior work on the topic:** According to the recent benchmark study *VideoGameQA-Bench* [45] by Taesiri et al., GPT-4o is the top-tier model for image-based game glitch detection, achieving an accuracy of 82.9% and an F1 score of 0.83 on the benchmark dataset. However, this dataset is constructed from publicly available data, which predominantly features clear and easily recognizable glitches. In contrast, real-world QA processes often involve subtle visual issues that may be difficult to spot during normal gameplay. To evaluate GPT-4o's performance under such industrial conditions, we randomly sampled 100 images with a similar distribution (50 normal gameplay images and 50 known-bug images). Each image was processed by GPT-4o using the game glitch detection prompt defined in *VideoGameQA-Bench*. When asked to classify whether an image contained a glitch, GPT-4o only achieved 46% accuracy and an F1 score of 0.52 on our data. These results indicate that, in a development setting, pure image-based detection struggles with the complexity of our data and underscores the need to provide additional input context. In contrast, our retrieval pipeline leverages both visual (frame) and textual (bug description) inputs, achieving Success@1 = 89% and F1@1 = 0.79.

**Comparison with naive baselines:** Section 5.2 shows that naive baselines perform poorly. To further investigate why, we examined the positional distribution of buggy keyframes. In a random sample of 50 videos, buggy keyframes appear at varied positions: the first keyframe is buggy in 15 videos, the last in 18, and the middle (or middle+1) in 30. This variability indicates that no simple keyframe-selection heuristic is effective for our data.

**Limitations of our approach:** Based on our results, we recommend applying automated bug frame retrieval primarily to categories with strong visual cues, such as Lighting & Shadow, Rendering & Texture, and UI & HUD. Categories like Animation & VFX or Performance are inherently temporal in nature and require fundamentally different approaches, such as multi-frame analysis or video-level modeling. Future work should explore incorporating temporal context into the pipeline to handle such cases.

**Practical Impact:** Our approach provides an efficient, scalable solution for reviewing gameplay videos. With a median length of 551 frames (~9 s), we achieve 0.89 Success@1, retrieving the correct buggy frame in nearly 90% of videos. This reduces manual effort from reviewing hundreds of frames to just one. Although the median videos are short, spotting the bug often requires multiple viewings, meaning the practical savings are even greater. All experiments cost under $100 (≈$0.054 per minute), showing that the method is both accurate and cost-effective.

## 7 Threats to Validity

***Internal validity:*** A potential threat to the internal validity of our results is the used encoding settings. We used the default FFmpeg settings to re-encode the videos. As these settings achieved a 98.79% bug coverage rate in our preliminary study, we did not fine tune these settings any further. However, there could be settings that result in an even higher bug coverage rate and a lower frame extraction rate. Future studies should further investigate the impact of encoding settings on our approach.

Another threat arises from the batch size used during MLLM analysis. Due to API limitations, we processed frames in batches for longer videos. If a bug-related sequence is split across batch boundaries or if the relevant context spans multiple batches, the MLLM may lack sufficient visual continuity to retrieve the correct frame. Despite its negligible impact on our studied videos, its effect on long video performance should be investigated by future studies.

Additional threats arise from components within our own pipeline. Specifically, we use GPT-4o to generate summaries of bug descriptions and assume its correctness. Although its outputs achieved 97% agreement with human judgment in a random 100-sample evaluation, this sample may not fully represent the diversity of bug types encountered in the data. As a result, the reliability of these summaries remains uncertain during full pipeline execution.

Furthermore, the prompting strategies used in the pipeline are straightforward, relying primarily on zero-shot or minimally guided prompts without task-specific fine-tuning. Although this design promotes ease of integration, it may limit performance in more ambiguous scenarios. Changes in prompt phrasing, structure, or context can influence the interpretation and ranking behavior of the MLLM and may produce different results.

Finally, manual evaluation of MLLM-generated ranked frames introduces potential bias. Determining whether a given frame accurately matches its bug description is sometimes subjective and ambiguous, particularly in difficult and subtle cases. These human judgments may inadvertently overestimate or underestimate the true performance of the model.

***External validity:*** A threat to the external validity is that the studied bug reports were all obtained from a single FPS (First Person Shooter) game. While the game features a variety of visual bug types and development workflows, the domain-specific nature of its mechanics, UI design, and bug reporting structure may not fully represent games in other genres. For example, the presentation of animation bugs, UI glitches, or logic errors may vary significantly between game types and game engines. Future studies should further investigate our approach for other types of games.

Another factor affecting external validity is the dependency on a specific version of the language model. Our pipeline is built entirely on the GPT-4o model, which is subject to updates, deprecation, or behavioral changes over time. The same prompts and inputs may yield different outputs in future versions or different models due to changes in model architecture, training data, backend configuration, etc. This model-specific dependency raises concerns about the long-term robustness of the approach, especially in production environments where model availability may change over time, causing performance variations.

## 8 Conclusion

We introduced an automated pipeline combining keyframe extraction and GPT-4o-based frame analysis to effectively retrieve bug frames from gameplay videos. Our pipeline demonstrates strong retrieval performance on real-world JIRA bug data from a popular FPS game, achieving an overall F1@1 score of 0.79 and a Success@1 of 0.89. We show particularly strong results in retrieving bug frames related to Lighting & Shadow, Physics & Collision, and UI & HUD.

We identified key limitations, particularly in Animation & VFX bugs, where the pipeline struggled mostly due to insufficient context captured by individual frames. Future work could enhance performance by incorporating temporal context analysis, fine-tuning prompting strategies, and evaluating our pipeline across diverse game genres and more extensive datasets. Overall, our approach presents an important, practical advancement in automated game QA, significantly reducing manual effort and providing robust support for visual bug retrieval in real-world game development scenarios.

**Acknowledgements:** This work was supported by Mitacs through the Mitacs Accelerate program (#IT42598: Visual Regression Testing of Video Games Using Foundation Models).

## References

[1] 2025. *FFmpeg*. https://ffmpeg.org/
[2] Anthropic. 2024. *Introducing the next generation of Claude*. https://www.anthropic.com/news/claude-3-family
[3] Sinan Ariyurek, Aysu Betin-Can, and Elif Surer. 2021. Automated Video Game Testing Using Synthetic and Humanlike Agents. *IEEE Transactions on Games* 13, 1 (2021), 50–67. doi:10.1109/TG.2019.2947597
[4] Atlassian. 2025. *Jira*. https://www.atlassian.com/software/jira
[5] Elham Azizi and Loutfouz Zaman. 2023. Automatic Bug Detection in Games using LSTM Networks. In *2023 IEEE Conference on Games (CoG)*. 1–4. doi:10.1109/CoG57401.2023.10333253
[6] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. 2008. What makes a good bug report?. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Atlanta, Georgia) *(SIGSOFT '08/FSE-16)*. Association for Computing Machinery, New York, NY, USA, 308–318. doi:10.1145/1453101.1453146
[7] Stefano Campanella, Emanuela Guglielmi, Rocco Oliveto, Gabriele Bavota, and Simone Scalabrino. 2024. Towards the Automatic Replication of Gameplays to Support Game Debugging. In *Proceedings of the 1st ACM International Workshop on Foundations of Applied Software Engineering for Games* (Porto de Galinhas, Brazil) *(FaSE4Games 2024)*. Association for Computing Machinery, New York, NY, USA, 1–6. doi:10.1145/3663532.3664465
[8] Norman Cliff. 1993. *Dominance Statistics: Ordinal Analyses to Answer Ordinal Questions*. Vol. 114. 494–509.
[9] Nathan Cooper, Carlos Bernal-Cárdenas, Oscar Chaparro, Kevin Moran, and Denys Poshyvanyk. 2021. It Takes Two to Tango: Combining Visual and Textual Information for Detecting Duplicate Video-Based Bug Reports. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 957–969. doi:10.1109/ICSE43902.2021.00091
[10] Yijiang River Dong, Tiancheng Hu, and Nigel Collier. 2024. Can LLM be a Personalized Judge? arXiv:2406.11657 [cs.CL] https://arxiv.org/abs/2406.11657
[11] EA. 2025. *EA Forum*. https://forums.ea.com/
[12] Sidong Feng and Chunyang Chen. 2022. GIFdroid: automated replay of visual bug reports for Android apps. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1045–1057. doi:10.1145/3510003.3510048
[13] Roberto Gallotta, Graham Todd, Marvin Zammit, Sam Earle, Antonios Liapis, Julian Togelius, and Georgios N. Yannakakis. 2024. Large Language Models and Games: A Survey and Roadmap. *IEEE Transactions on Games* (2024), 1–18. doi:10.1109/TG.2024.3461510
[14] Google. 2024. *Introducing Gemini 2.0: our new AI model for the agentic era*. https://blog.google/technology/google-deepmind/google-gemini-ai-update-december-2024/
[15] Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, Shengjie Ma, Honghao Liu, Saizhuo Wang, Kun Zhang, Yuanzhuo Wang, Wen Gao, Lionel Ni, and Jian Guo. 2025. A Survey on LLM-as-a-Judge. arXiv:2411.15594 [cs.CL] https://arxiv.org/abs/2411.15594
[16] Emanuela Guglielmi, Gabriele Bavota, Rocco Oliveto, and Simone Scalabrino. 2025. Automatic Identification of Game Stuttering via Gameplay Videos Analysis. *ACM Trans. Softw. Eng. Methodol.* 34, 2, Article 38 (Jan. 2025), 29 pages. doi:10.1145/3695992
[17] Emanuela Guglielmi, Simone Scalabrino, Gabriele Bavota, and Rocco Oliveto. 2022. Towards Using Gameplay Videos for Detecting Issues in Video Games. arXiv:2204.04182 [cs.SE] https://arxiv.org/abs/2204.04182
[18] Emanuela Guglielmi, Simone Scalabrino, Gabriele Bavota, and Rocco Oliveto. 2023. Using gameplay videos for detecting issues in video games. *Empirical Softw. Engg.* 28, 6 (Oct. 2023), 32 pages. doi:10.1007/s10664-023-10365-0
[19] Sihao Hu, Tiansheng Huang, Gaowen Liu, Ramana Rao Kompella, Fatih Ilhan, Selim Furkan Tekin, Yichang Xu, Zachary Yahn, and Ling Liu. 2025. A Survey on Large Language Model-Based Game Agents. arXiv:2404.02039 [cs.AI] https://arxiv.org/abs/2404.02039
[20] Sumandeep Kaur, Lakhwinder Kaur, and Madan Lal. 2024. An effective Key Frame Extraction technique based on Feature Fusion and Fuzzy-C means clustering with Artificial Hummingbird. *Scientific Reports* 14 (11 2024). doi:10.1038/s41598-024-75923-y
[21] Eugene Klishevich, Yegor Denisov-Blanch, Simon Obstbaum, Igor Ciobanu, and Michal Kosinski. 2025. Measuring Determinism in Large Language Models for Software Code Review. arXiv:2502.20747 [cs.SE] https://arxiv.org/abs/2502.20747
[22] Vasilii Korolkov. 2025. Scene Detection Policies and Keyframe Extraction Strategies for Large-Scale Video Analysis. arXiv:2506.00667 [cs.CV] https://arxiv.org/abs/2506.00667
[23] Bo Li, Yuanhan Zhang, Dong Guo, Renrui Zhang, Feng Li, Hao Zhang, Kaichen Zhang, Yanwei Li, Ziwei Liu, and Chunyuan Li. 2024. LLaVA-OneVision: Easy Visual Task Transfer. *arXiv preprint arXiv:2408.03326* (2024).
[24] Haitao Li, Qian Dong, Junjie Chen, Huixue Su, Yujia Zhou, Qingyao Ai, Ziyi Ye, and Yiqun Liu. 2024. LLMs-as-Judges: A Comprehensive Survey on LLM-based

Evaluation Methods. arXiv:2412.05579 [cs.CL] https://arxiv.org/abs/2412.05579
[25] Dayi Lin, Cor-Paul Bezemer, and Ahmed E. Hassan. 2019. Identifying gameplay videos that exhibit bugs in computer games. *Empirical Software Engineering* (12 2019). doi:10.1007/s10664-019-09733-6
[26] Charles Lovering, Michael Krumdick, Viet Dac Lai, Seth Ebner, Nilesh Kumar, Varshini Reddy, Rik Koncel-Kedziorski, and Chris Tanner. 2025. Language Model Probabilities are Not Calibrated in Numeric Contexts. arXiv:2410.16007 [cs.AI] https://arxiv.org/abs/2410.16007
[27] Wentao Lu, Alexander Senchenko, Abram Hindle, and Cor-Paul Bezemer. 2025. *Automated Bug Frame Retrieval from Gameplay Videos Using Vision-Language Models Replication Package.* doi:10.5281/zenodo.17203525
[28] Weiyu Ma, Qirui Mi, Yongcheng Zeng, Xue Yan, Yuqiao Wu, Runji Lin, Haifeng Zhang, and Jun Wang. 2024. Large Language Models Play StarCraft II:Benchmarks and A Chain of Summarization Approach. In *Advances in Neural Information Processing Systems*, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (Eds.), Vol. 37. Curran Associates, Inc., 133386–133442. https://proceedings.neurips.cc/paper_files/paper/2024/file/f0ebc318e2df08360b2df559e81602e5-Paper-Conference.pdf
[29] David Melhart, Matthew Barthet, and Georgios N. Yannakakis. 2025. Can Large Language Models Capture Video Game Engagement? arXiv:2502.04379 [cs.CV] https://arxiv.org/abs/2502.04379
[30] Microsoft. 2025. *Azure.* https://azure.microsoft.com/en-ca
[31] M. Jehanzeb Mirza, Leonid Karlinsky, Wei Lin, Sivan Doveh, Jakub Micorek, Mateusz Kozinski, Hilde Kuehne, and Horst Possegger. 2025. Meta-prompting for Automating Zero-Shot Visual Recognition with LLMs. In *Computer Vision – ECCV 2024*, Aleš Leonardis, Elisa Ricci, Stefan Roth, Olga Russakovsky, Torsten Sattler, and Gül Varol (Eds.). Springer Nature Switzerland, Cham, 370–387.
[32] Mistral. 2025. *Frontier AI. In Your Hands.* https://mistral.ai/
[33] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2015. Auto-completing bug reports for Android applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 673–686. doi:10.1145/2786805.2786857
[34] Aishik Nagar, Shantanu Jaiswal, and Cheston Tan. 2024. Zero-Shot Visual Reasoning by Vision-Language Models: Benchmarking and Analysis. In *2024 International Joint Conference on Neural Networks (IJCNN)*. 1–8. doi:10.1109/IJCNN60899.2024.10650020
[35] OpenAI. 2024. *GPT-4o mini: advancing cost-efficient intelligence.* https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/
[36] OpenAI. 2024. *Hello GPT-4o.* https://openai.com/index/hello-gpt-4o/
[37] OpenAI. 2024. *OpenAI Platform.* https://platform.openai.com/docs/api-reference/chat/create
[38] Ciprian Paduraru, Miruna Paduraru, and Alin Stefanescu. 2021. Automated game testing using computer vision methods. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*. 65–72. doi:10.1109/ASEW52652.2021.00024
[39] Max Peeperkorn, Tom Kouwenhoven, Dan Brown, and Anna Jordanous. 2024. Is Temperature the Creativity Parameter of Large Language Models? arXiv:2405.00492 [cs.CL] https://arxiv.org/abs/2405.00492
[40] Reddit. 2022. *Visual bug when Wraith finisher gets interupted.* https://www.reddit.com/r/apexlegends/comments/13i8gmp/visual_bug_when_wraith_finisher_gets_interupted
[41] Iain E. Richardson. 2010. *The H.264 Advanced Video Compression Standard, Second Edition.* Wiley.
[42] Alexander Senchenko, Naomi Patterson, Hamman Samuel, and Dan Ispir. 2022. SUPERNOVA: Automating Test Selection and Defect Prevention in AAA Video Games Using Risk Based Testing and Machine Learning. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 345–354. doi:10.1109/icst53961.2022.00043
[43] Mohammad Reza Taesiri and Cor-Paul Bezemer. 2025. VIDEOGAMEBUNNY: Towards vision assistants for video games. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision* (2025-03-01).
[44] Mohammad Reza Taesiri, Tianjun Feng, Cor-Paul Bezemer, and Anh Nguyen. 2024. GlitchBench: Can Large Multimodal Models Detect Video Game Glitches?. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 22444–22455.
[45] Mohammad Reza Taesiri, Abhijay Ghildyal, Saman Zadtootaghaj, Nabajeet Barman, and Cor-Paul Bezemer. 2025. VideoGameQA-Bench: Evaluating Vision-Language Models for Video Game Quality Assurance. arXiv:2505.15952 [cs.CV] https://arxiv.org/abs/2505.15952
[46] Mohammad Reza Taesiri, Finlay Macklon, and Cor-Paul Bezemer. 2022. CLIP meets GamePhysics: towards bug identification in gameplay videos using zero-shot transfer learning. In *Proceedings of the 19th International Conference on Mining Software Repositories* (Pittsburgh, Pennsylvania) *(MSR '22)*. Association for Computing Machinery, New York, NY, USA, 270–281. doi:10.1145/3524842.3528438
[47] Xuchen Tan, Deenu Yadav, Faiz Ahmed, and Maleknaz Nayebi. 2025. ImageR: Enhancing Bug Report Clarity by Screenshots. arXiv:2505.01925 [cs.SE] https://arxiv.org/abs/2505.01925
[48] Andrew Truelove, Shiyue Rong, Eduardo Santana de Almeida, and Iftekhar Ahmed. 2023. Finding the Needle in a Haystack: Detecting Bug Occurrences in Gameplay Videos. arXiv:2311.10926 [cs.SE] https://arxiv.org/abs/2311.10926
[49] Alicia Vidler and Toby Walsh. 2025. Playing games with Large language models: Randomness and strategy. arXiv:2503.02582 [cs.AI] https://arxiv.org/abs/2503.02582
[50] Dingbang Wang, Zhaoxu Zhang, Sidong Feng, William G. J. Halfond, and Tingting Yu. 2025. An Empirical Study on Leveraging Images in Automated Bug Report Reproduction. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*. 27–38. doi:10.1109/MSR66628.2025.00019
[51] Frank Wilcoxon. 1992. *Individual Comparisons by Ranking Methods.* Springer New York, New York, NY, 196–202. doi:10.1007/978-1-4612-4380-9_16
[52] Yanfu Yan, Nathan Cooper, Oscar Chaparro, Kevin Moran, and Denys Poshyvanyk. 2024. Semantic GUI Scene Learning and Video Alignment for Detecting Duplicate Video-based Bug Reports. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 232, 13 pages. doi:10.1145/3597503.3639163
[53] Yan Zhao, Weihao Zhang, Enyi Tang, Haipeng Cai, Xi Guo, and Na Meng. 2021. A Lightweight Approach of Human-Like Playtesting. arXiv:2102.13026 [cs.SE] https://arxiv.org/abs/2102.13026
[54] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, Hao Zhang, Joseph E Gonzalez, and Ion Stoica. 2023. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. In *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., 46595–46623. https://proceedings.neurips.cc/paper_files/paper/2023/file/91f18a1287b398d378ef22505bf41832-Paper-Datasets_and_Benchmarks.pdf