GameLab: AI-Enabled Cloud Gaming Testbed

Haseeb Ur Rehman
University of Ottawa
Ottawa, Canada

Shervin Shirmohammadi
University of Ottawa
Ottawa, Canada

Ihab Amer
American University of Sharjah
Sharjah, UAE

Mohamed Hefeeda
Simon Fraser University
Burnaby, Canada

Abstract

We present GameLab, an open-source, AI-enabled cloud gaming testbed built on WebRTC. Unlike existing open-source stacks and deployment-oriented pipelines (e.g., GamingAnywhere, Sunshine/-Moonlight, and Unity Render Streaming), GameLab is designed for *AI-in-the-loop* systems research: it provides programmable interfaces on both the server and client, with well-defined hook points to plug in machine learning modules on demand (e.g., super-resolution, denoising, object detection, QoE estimation, and learned rate control). GameLab also collects detailed transport and application traces for online and offline analyses of gaming sessions and network behavior. To enable reliable *objective* evaluation in interactive settings, GameLab embeds compact QR-based frame identifiers into the video stream, allowing accurate computation of full-reference quality metrics, such as PSNR, SSIM, and VMAF, even under frame loss, reordering, and duplication. Finally, GameLab supports GPU-based visualization of frames and model outputs for interactive inspection without costly GPU-to-CPU transfers.

We demonstrate the versatility of GameLab through multiple end-to-end experiments, including: (i) transparently applying video upscalers to reduce bandwidth, (ii) deploying multiple client-side object detection models, and (iii) implementing a server-side visual-complexity-driven bitrate ladder for rate adaptation.
Github: https://github.com/haseebfazal/ai-testbed.git.

CCS Concepts

• **Networks** → **Cloud computing**; **Network experimentation**; **Network performance analysis**; **Network measurement**.

Keywords

Cloud Gaming, WebRTC

ACM Reference Format:
Haseeb Ur Rehman, Shervin Shirmohammadi, Ihab Amer, and Mohamed Hefeeda. 2026. GameLab: AI-Enabled Cloud Gaming Testbed. In *ACM Multimedia Systems Conference 2026 (MMSys '26), April 04–08, 2026, Hong Kong, Hong Kong*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3793853.3799802

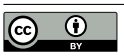
1 Introduction

Cloud gaming (CG) renders interactive game scenes on a remote machine and streams the resulting video to a thin client while the client transmits user inputs back to the server. This architecture removes local hardware constraints but makes the end-to-end experience highly sensitive to bandwidth, latency, jitter, and server-side compute cost. Consequently, a growing body of systems work has been exploring the utilization of various AI modules inside the CG pipeline, including learned/content-aware encoding and adaptation [6, 12, 15], perceptual quality and QoE/KQI prediction [24, 22, 5, 4], neural enhancement such as super-resolution (SR) [18, 9], and content-aware processing based on objects/regions-of-interest [7].

Evaluating AI ideas in a *reproducible* end-to-end setup, however, remains challenging: researchers must implement and integrate many components, including real-time capturing of game frames, video encoding, adaptive transmission, video decoding, and player input control, while also logging network and application metrics and keeping the system modifiable enough to rapidly prototype new AI models. Existing CG/game-streaming stacks and deployment-oriented pipelines are typically not designed for *AI-in-the-loop* systems research. For example, GamingAnywhere [8] provides valuable baselines, but it is difficult to extend and instrument for modern GPU-accelerated inference modules. End-user solutions such as Sunshine+Moonlight [10, 11] prioritize practical gameplay deployment over fine-grained research control and programmable hook points. Engine-specific pipelines like Unity Render Streaming target only Unity applications and deliver frames to a WebRTC browser client [19, 21, 20]. This requires access to the Unity project/source code, which limits evaluation on arbitrary commercial games and complicates reusing native accelerated PyTorch/TensorFlow inference codepaths, pushing experimentation toward browser-oriented AI stacks [17, 23, 16].

Moreover, even when a streaming pipeline is available, *objective* evaluation is fragile in interactive settings because loss, reordering, and frame duplication can break timestamp-based alignment between received frames and server-side ground truth, complicating the computation of full-reference video quality metrics, e.g., PSNR, SSIM, and VMAF, in diverse network conditions. Such metrics are crucial for evaluating new ideas and conducting systems research.

This paper presents an **open-source, Python-first, AI-enabled cloud gaming testbed** that provides end-to-end control of the CG pipeline. The testbed, called GameLab, is built on `aiortc` [1] and exposes practical **hook points** for server-side pre-encode and client-side post-decode processing, enabling seamless integration of arbitrary PyTorch/TensorFlow modules (e.g., SR, denoising, object detection, QoE predictions, and learned rate controllers) without re-architecting the system or leaving Python. To make *objective* evaluation reproducible under packet loss/duplication, we introduce a **QR-based frame identity and alignment mechanism**: the server overlays a compact QR code encoding a unique frame ID, and the client decodes this code to match each received frame to its server-side reference, enabling repeatable PSNR/SSIM/VMAF computation across bandwidth sweeps and real traces. For efficient



MMSys '26, Hong Kong, Hong Kong

ACM ISBN 979-8-4007-2481-7/2026/04
https://doi.org/10.1145/3793853.3799802

experimentation and profiling, the testbed supports GPU-based tensor visualization via `cudacanvas` [13], integrates a lightweight video complexity backend [2], and includes supporting components such as TCP input channel for keyboard/mouse replay, unified runtime logging (bitrate, RTT/jitter/loss, FPS, and module runtimes), and reference plug-ins demonstrating extensibility (upsampling, object detection, and complexity-driven rate adaptation hooks).

We make the following contributions:

- An open-source, Python-first CG testbed that exposes server- and client-side hooks for integrating AI modules in real time.
- A QR-based measurement pipeline that enables reliable frame alignment and repeatable full-reference quality evaluation (PSNR/SSIM/VMAF) despite packet loss, reordering, and frame duplication.
- Reference plug-ins (Upscaling, object detection, complexity analysis) and reproducibility assets (scripts/configs/ log formats) to accelerate future research on AI-driven CG systems.

2 Related Work

2.1 Current Cloud Gaming Testbeds and Stacks

Several CG / game-streaming stacks and deployment-oriented pipelines exist and provide valuable baselines, but they are not designed for rapid *AI-in-the-loop* systems research.

Academic/open stacks. GamingAnywhere [8] is a widely-cited open CG system and has served as a baseline in many research prototypes. However, it primarily targets end-to-end game streaming rather than *programmable* experimentation, and its publicly released codebase reflects a legacy toolchain (e.g., the latest release is v0.8.0 from Jan. 2015, and the Windows build targets Win32 using Visual Studio 2010 and the June 2010 DirectX SDK), making it unsuitable for modern AI-in-the-loop workflows (hook points for model inference, fine-grained instrumentation/logging, and tight integration with contemporary GPU runtimes and Python training stacks).

Community streaming ecosystems. Sunshine (host) + Moonlight (client) [10, 11] have matured into practical low-latency remote-play solutions. These systems prioritize real-world usability and performance rather than research extensibility: they generally do not expose research-grade interfaces for inserting, benchmarking, and swapping inference modules (SR/denoising/detection) at different stages of the pipeline, nor do they provide standardized ground-truth alignment mechanisms for full-reference quality evaluation in realistic network conditions, where packets can be lost and reordered.

Engine-specific pipelines. Unity Render Streaming [19, 21, 20] streamlines Unity deployment by delivering frames to a WebRTC browser receiver, which is convenient for demos but can hinder reuse of native accelerated PyTorch/TensorFlow inference codepaths (e.g., CUDA/ROCm/Metal) and instead push experimentation toward browser-oriented AI stacks such as TensorFlow.js over WebGL/WebGPU [17, 23, 16]. Moreover, because it targets *Unity applications*, experiments typically require access to a Unity project (or the ability to modify/rebuild the application), which limits evaluation on closed-source or non-Unity commercial games. As a result, Unity/browser-centric pipelines are less suitable for systems research that needs flexible native-GPU hook points and precise cross-client measurement.

2.2 AI in Cloud Gaming Pipelines

A growing body of systems work is exploring learned components for improving efficiency, robustness, and perceived quality in CG and interactive video, motivating the need for a flexible testbed that can *plug in* such modules without re-architecting the full system.

Neural enhancement (SR) in the delivery loop. Neural enhancement has been explored both as a systems component and as an architectural choice: LevelUp demonstrates a thin-cloud game live streaming design where edge resources can transcode and enhance video quality (including SR) under bandwidth constraints [3].

Perceptual quality prediction and measurement. Gaming-oriented perceptual quality modeling spans perceptual-dimension-based estimators [24], no-reference deep models such as NDNetGaming [22], and mobile cloud-gaming predictors such as GAMIVAL [5], alongside AI-based estimation of end-to-end Key Quality Indicators (KQIs) from network-side metrics [4]. Collectively, these works motivate testbeds that support controlled bandwidth/latency/jitter experiments and reliable alignment for computing objective full-reference metrics when ground truth is available.

Learned encoding, adaptation, and content-aware processing. Content-aware encoding for CG includes region-/saliency-aware bit allocation (e.g., CAVE [6]), learned ROI prediction to save bandwidth while preserving perceptual quality (e.g., DeepGame [12]), and newer content-adaptive schemes for *interactive game streaming* under strict latency/compute constraints [15]. Complementary work exploits game semantics via selective object encoding to reduce bitrate with limited QoE impact [7]. Overall, these directions motivate CG pipelines with clean insertion points for AI vision models and learned controllers.

Positioning of GameLab. In contrast to the above systems, which often optimize a *single* component within a tightly-coupled stack, our goal is to provide an open CG testbed that supports rapid AI-in-the-loop prototyping (via server-side and client-side hooks) *and* robust, repeatable objective evaluation in real network environments (where packet loss/reordering/duplication occur), enabling researchers to assess and compare AI modules (e.g., RL-based rate-controllers, SR, denoising, object detection, QoE predictors, learned controllers) on equal footing.

3 GameLab Architecture

Figure 1 provides an overview of GameLab, an open-source, AI-enabled cloud gaming testbed. GameLab follows the standard CG loop (input → render/capture → encode/transport → decode/display) while adding two capabilities that are difficult to obtain in existing CG stacks: (i) *programmable hook points* (server-side pre-encode and client-side post-decode) for rapidly inserting AI modules without re-architecting the system, and (ii) *robust measurement* via QR-based frame identity to enable reliable full-reference evaluation under real transport artifacts (loss, reordering, duplication). GameLab runs two parallel paths: a lightweight TCP control channel that relays keyboard/mouse events, and a WebRTC media channel that streams encoded frames to the client.

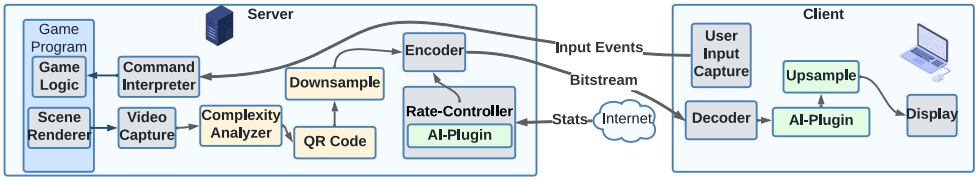


Figure 1: Design of GameLab. In addition to streaming gaming sessions, GameLab provides two Python hook points: (i) server *pre-encode* (for frame analysis/modification and QR frame IDs for full-reference quality metrics) and (ii) client *post-decode* (for pluggable AI modules such as SR and object detection). It also offers detailed logs for offline and online analyses.

3.1 Server-side Pipeline

The sender (`server.py`) is implemented in Python on top of `aiortc` [1]. The server executes the game locally and captures the rendered output using the MSS screen-grabbing library [14]. In our implementation, captured frames are converted to BGR arrays (OpenCV-native layout, avoiding extra color-space conversions and enabling fast OpenCV/Numpy ops) before encoding.

We use OS-level screen capture (not engine/frame-buffer hooks) to support unmodified commercial games across diverse runtimes and to avoid anti-cheat/integrity restrictions on injection. While it may add modest overhead, it is broadly applicable and reproducible across games and setups.

Video capture. A custom (`CaptureScreen`) grabs a centered region of a selected monitor at a configurable resolution and FPS. This stage is intentionally kept simple and scriptable, so researchers can replace the capture backend or add pre-processing without touching the transport stack.

Pre-encode hook points: complexity analysis, downsampling, and QR overlay. Before encoding, GameLab exposes a generic pre-encode hook point where researchers can plug in AI modules and instrumentation. We provide the following reference modules corresponding to Figure 1:

Optional server-side instrumentation. *Frame complexity analysis* estimates the spatial detail of each game-rendered frame (and thus its compression/upscaling difficulty), which can drive content-aware adaptation, learned control, and offline labeling. As a strong real-time baseline, we instantiate this hook with EVCA [2], a state-of-the-art real-time complexity estimator; the design is modular and can be replaced with any other real-time analyzer. This hook is implemented in `server.py` (`ComplexityAnalyzer`) and reproduces EVCA's blockwise DCT energy scoring on the luminance plane using `torch_dct` and the same block size and weighting mask as the original, integrated as a native PyTorch module to avoid subprocess overhead; scores are logged per-frame or averaged over a GOP window (`-enable-complexity`, `-gop-len`).

Downsampling before encode reduces transmitted pixel rate (e.g., ×2/×3) to lower bandwidth demand and often latency; we apply it early so complexity/QR instrumentation matches the transmitted resolution (`server.py`, enabled via `-downsample`). When downsampling is active, metrics are computed by comparing the post-decode (optionally upsampled) client output against the server-side pre-encode reference upscaled to display resolution, so PSNR/SSIM/VMAF reflects end-to-end quality including both codec and upsampler effects.

QR-based frame identity robustly aligns received frames with server-side references under timestamp drift, loss, and duplication by overlaying a compact QR code encoding frame ID and metadata (index, timestamp, resolution), enabling reliable full-reference metrics and debugging. It is implemented in `server.py` (`QualityLogger`) as a small corner patch (default top-right) with configurable size and margin; by default we use a 160×160 px patch (about 1.2% of a 1920×1080 frame) and scale it with resolution to limit occlusion, and the QR region can be masked/cropped during metric computation if strict visual fidelity is required (`-quality-logging`, `-qr-size`, `-qr-corner`). Importantly, the QR mechanism does *not* recover lost frames; it is a purely offline alignment tool that matches each received frame to its server-side reference, skipping lost frames, without modifying transport behavior.

These modules serve as reference implementations; the same pre-encode hook is generic and can host arbitrary processing (e.g., denoising, filtering, feature extraction, overlays) without changing the rest of the pipeline. The server also exposes WebRTC transport statistics (e.g., bitrate/RTT/jitter) to support pluggable rate control: by default it uses WebRTC congestion control, but users can implement heuristic or learned (including RL) controllers on top of these signals.

Encoding, transport, and network stats. After pre-encode processing, frames are wrapped as `VideoFrame` (`bgr24`) and delivered through the WebRTC pipeline. We use a modified fork of `aiortc` that (i) adds transport/statistics logging used by our measurement scripts, (ii) increases the default jitter buffer capacity to sustain high-bitrate game streams under transient delay variation, and (iii) exposes a server-side rate-control API that allows external controllers to program the encoder target bitrate at runtime. The fork also exposes optional hardware-accelerated encode/decode (e.g., NVENC/CUVID) when `PyAV` is built with CUDA support, while the default remains software H.264 for portability. We chose `aiortc` for its Python-native, asyncio-based design that integrates directly with PyTorch/TensorFlow without language-boundary overhead; its single-process concurrency is not a bottleneck for single-session research experiments. Our fork uses GCC with REMB-based feedback; TWCC and explicit pacing are not currently implemented. The sender periodically queries WebRTC statistics and logs effective send bitrate via `RTCOutboundRtpStreamStats` (computed from `bytesSent` over time); these signals are consumed by rate-control plug-ins (Figure 1). We default to CPU encode/decode since CPU↔GPU transfer overhead can offset hardware-encoder speedups in our measurements.

Shared-memory rate-control hook. Since setting a fixed bitrate in `aiortc` does not reliably override GCC, we provide an optional shared-memory interface: the pre-encode stage writes a 32-bit bitrate value (bps) into a named region; the streaming process polls and applies it to the encoder. This lets controllers combine raw-frame signals (e.g., EVCA complexity) with transport metrics (RTT/loss/jitter) for anything from heuristics to RL controllers, without re-plumbing WebRTC. When disabled, the system falls back to GCC.

3.2 Client-side Pipeline

The receiver (`client.py`) terminates the WebRTC session, decodes video frames, and applies post-decode processing prior to display and/or saving.

Post-decode AI plug-in interface. Decoded frames are converted to a GPU tensor (`torch.Tensor`, CHW, float $[0, 1]$), making it straightforward to insert any PyTorch/TensorFlow module (SR, denoising, segmentation, QoE prediction) without leaving Python.

Built-in examples: upsampling and object detection. To demonstrate the plug-in workflow, the client includes: (i) an optional real-time upsampling path using `torch.nn.functional.interpolate` (nearest/bilinear/bicubic) as a baseline, and (ii) an optional GPU object detection module using a TorchVision SSDLite MobileNetV3 detector, executed every N frames with configurable thresholding and a top-K cap for overlay rendering. These examples serve as reference implementations for integrating heavier AI modules.

GPU-native visualization and optional frame dumping. For interactive experimentation, the client displays tensors directly using `cudacanvas` [13], avoiding GPU→CPU copies that can distort latency measurements. Optionally, the receiver can dump processed frames to disk (PNG) for offline evaluation; combined with the server's QR-based frame identity, this enables robust alignment for full-reference metrics (PSNR/SSIM/VMAF) across bandwidth sweeps and real network traces.

Client transport statistics. The client also logs receive-side transport bitrate using `RTCTransportStats` (`bytesReceived`), enabling correlation of network dynamics with post-decode AI module behavior and observed visual quality.

4 Demonstration and Evaluation of GameLab

This section demonstrates how GameLab enables *AI-in-the-loop* cloud-gaming experiments through (i) programmable server/client hook points, (ii) robust measurement via QR-based frame identity, and (iii) unified runtime trace collection. Our goal is to make it easy to *swap* AI modules and *reproduce* end-to-end sessions while collecting the signals needed for online control and offline analysis.

4.1 Configuring and Running GameLab

GameLab provides scriptable entry points for both endpoints: `Server/server.sh` launches the WebRTC sender (and the C++ input server), and `Client/client.sh` launches the WebRTC receiver (and the input client). Researchers can enable/disable each capability by editing a small set of flags in these scripts (e.g., `DOWNSAMPLE`, `ENABLE_QUALITY_LOGGING`, `ENABLE_COMPLEXITY`, `ENABLE_EVCA_BITRATE` on the server; `UPSAMPLE`, `UPSAMPLE_MODE`, `DETECT`, `SAVE_FRAMES` on the client).

Reliable full-reference evaluation via QR identity. When `ENABLE_QUALITY_LOGGING=1`, the server overlays a compact QR code per frame encoding a unique identifier and metadata. If `SAVE_FRAMES=1` is enabled on the client, the decoded video frames can be paired with their corresponding server-side references by decoding the QR IDs offline, enabling robust PSNR/SSIM/VMAF computation even under real-time artifacts such as loss, duplication, and timestamp drift.

Trace collection. Both endpoints log time-stamped transport and application signals (e.g., achieved bitrate and WebRTC statistics). These traces can be redirected to files for offline analysis, or used online by external control logic (e.g., shared-memory rate controllers).

4.2 Use Case 1: Swappable Video Upsamplers

We demonstrate how GameLab supports *end-to-end* evaluation while swapping receiver-side enhancement modules without changing capture, encoding, or transport. In our runs, we compare three representative pipelines under identical streaming conditions: (i) **native 1080p streaming**, (ii) **server-side downsampled streaming** with a **client-side interpolation baseline** (nearest/bilinear/bicubic via `UPSAMPLE_MODE`), and (iii) the same post-decode hook populated with a **neural upsampler**, instantiated here with RT4KSR [25]. Figures 2a–2d summarize the resulting SSIM/VMAF behavior across the sweep using QR-aligned frame pairs.

On the client, decoded WebRTC frames are converted into a tensor-first representation and passed through the post-decode hook. This makes the enhancement module a drop-in component: replacing bicubic with RT4KSR (or *any* other PyTorch/TensorFlow SR model) only changes the function applied at this hook point, without affecting the rest of the streaming loop. Concretely, in our implementation the insertion point is:

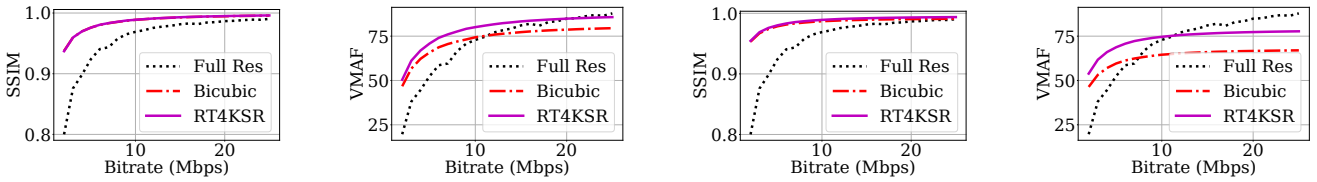
```
img_chw = TF.to_tensor(rgb).to(self.device).clamp(0, 1)
if self.upsample:
    img_chw = upsample_chw(img_chw, scale, self.upsample_mode)  #
    ↪  swap SR here
```

This capability is hard to reproduce in many existing pipelines because swapping post-decode AI modules while keeping the rest of the streaming loop fixed—and still obtaining reliable full-reference metrics via robust frame alignment—typically requires substantial re-engineering and custom instrumentation.

Network traces alongside RD curves. In addition to full-reference quality, GameLab logs WebRTC transport statistics during each run. Figure 3 shows representative RTT and jitter traces captured during the same sessions, illustrating how researchers can jointly analyze time-varying network dynamics together with objective quality and AI-module choices.

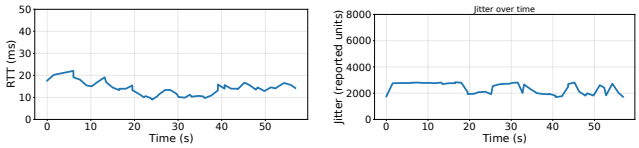
4.3 Use Case 2: Client-side Analytics

Next, we demonstrate that GameLab can host real-time client analytics at the post-decode hook point. The repository includes an example object-detection plug-in (TorchVision SSDLite MobileNetV3)



(a) CS2 (SSIM) 540p → 1080p (b) CS2 (VMAF) 540p → 1080p (c) CS2 (SSIM) 360p → 1080p (d) CS2 (VMAF) 360p → 1080p

Figure 2: CS2 bandwidth sweep: SSIM/VMAF vs. achieved bitrate for three pipelines (native 1080p, interpolation baseline, and neural upsampler at the same post-decode hook). Left: X2 (540p→1080p). Right: X3 (360p→1080p). Metrics use QR-aligned frames.



(a) RTT over time (b) Jitter over time

Figure 3: RTT and jitter traces logged from WebRTC transport statistics during the CS2 bandwidth-sweep runs.

that can be enabled via `Client/client.sh` (`DETECT=1`) and configured by frequency (`DETECT_EVERY`) and thresholds (`DETECT_THRESH`, `DETECT_TOPK`). Figure 4 shows sample annotated frames from two games (CS2 and OW2), illustrating how semantic signals (detections/ROIs) can be produced online for downstream uses such as ROI-aware encoding, selective enhancement, or gameplay-aware telemetry. In contrast to stacks that primarily support end-to-end streaming but provide limited programmable insertion points, GameLab makes client-side AI analytics a first-class, toggleable component that can be evaluated under the same transport conditions and logging interface.

4.4 Use Case 3: Complexity-driven Bitrate Ladder

Finally, we demonstrate GameLab's content-aware control loop by enabling the server-side EVCA complexity analyzer and the shared-memory rate-control hook (`ENABLE_COMPLEXITY=1`, `ENABLE_EVCA_BITRATE=1` in `Server/server.sh`). The included example maps the online spatial complexity score (SC) to a simple bitrate ladder (three buckets), producing a piecewise-constant target bitrate signal that can be consumed by the sender at runtime.

```
# ... inside CaptureScreen.recv(), after computing GOP-average
↪  complexity (sc_avg)
if self._bitrate_ctl is not None:
    target_bps, desired_bps, gcc_bps =
    ↪  self._bitrate_ctl.update_from_complexity(sc_avg)  # swap
    ↪  rate controller here
```

Complexity–bitrate coupling. Figure 5 shows the time series of (left) EVCA complexity (SC) and (right) the selected target bitrate. We overlay the ladder thresholds to make bucket transitions explicit; whenever SC crosses a threshold, the controller switches to the corresponding rung (e.g., 8/12/16 Mbps). This example is intentionally simple and is meant to demonstrate GameLab's programmability: richer heuristics (hysteresis, smoothing, minimum dwell time) or learned controllers can replace the ladder without changing the system architecture. Many prior systems expose only the default congestion-control behavior as a black box; GameLab instead makes the sender-side control loop *programmable*, so researchers can rapidly prototype and compare heuristic or learned controllers using the same signals and logs.

4.5 Reproducibility and Overheads

Why these demonstrations matter. GameLab is designed for *controlled, repeatable AI-in-the-loop experiments* via: (i) programmable hook points, (ii) unified logging, and (iii) QR-based frame identity under loss/reordering/duplication. PSNR, SSIM, and VMAF serve as reproducible proxies; subjective evaluation may complement them for certain AI modules. The following are *feature demonstrations* rather than absolute-performance comparisons.

Reproducibility. Script-controlled flags and time-stamped logs at both endpoints make it straightforward to rerun sessions while toggling a single capability (QR on/off, downsample factor, post-decode plug-ins, or rate-control hook), supporting fair module comparisons under consistent conditions.

Overheads. The QR identity mechanism is designed to be lightweight (small overlay region per frame) while substantially improving robustness of offline alignment. Similarly, the EVCA analyzer is optional and configurable (block size and GOP averaging window). For interactive experimentation, the client can display frames via GPU-native visualization to avoid extra device transfers that would otherwise distort latency measurements.

5 Conclusion

We release an open-source, Python-first AI-enabled cloud gaming testbed built on `aiortc` with server pre-encode and client post-decode hooks for plug-in ML modules (e.g., SR and analytics). For repeatable evaluation, we overlay QR-based frame IDs to robustly align received and reference frames and compute full-reference metrics (PSNR/SSIM/VMAF) under loss/duplication; we note these metrics serve as reproducible proxies and subjective evaluation may complement them for certain AI modules. We demonstrate reproducible end-to-end studies including bandwidth sweeps with

(a) CS2 example 1 (b) CS2 example 2 (c) OW2 example 1 (d) OW2 example 2

Figure 4: Client-side object detection as a post-decode plug-in. Annotated frames from CS2 and OW2 showing detector outputs computed on the client after WebRTC decoding.

(a) EVCA complexity over time (b) selected bitrate over time

Figure 5: Time-series behavior of EVCA during a streaming run: (left) the estimated spatial complexity (EVCA SC), and (right) the corresponding target bitrate selected from the ladder.

swappable upsamplers (bicubic vs. RT4KSR), client-side object detection, and server-side spatial complexity tracing with transport logging.

References

[1] aiortc contributors. 2025. Aiortc: webrtc for python. (2025). https://github.com/aiortc/aiortc.

[2] Hadi Amirpour, Mohammad Ghasempour, Lingfeng Qu, Wassim Hamidouche, and Christian Timmerer. 2024. Evca: enhanced video complexity analyzer. In *Proc. ACM MMSys '24*, 285–291. doi:10.1145/3625468.3652171.

[3] Lixiang Ao and Landon P. Cox. 2020. LevelUp: thin-cloud game livestreaming. (2020). https://www.microsoft.com/en-us/research/wp-content/uploads/2020/08/levelup-sec20.pdf.

[4] Carlos Baena, O. S. Peñaherrera-Pulla, Raquel Barco, and Sergio Fortes. 2023. Measuring and estimating key quality indicators in cloud gaming services. *Computer Networks*, 231, 109808. doi:10.1016/j.comnet.2023.109808.

[5] Yu-Chih Chen, Avinab Saha, Chase Davis, Bo Qiu, Xiaoming Wang, Rahul Gowda, Ioannis Katsavounidis, and Alan C. Bovik. 2023. GAMIVAL: video quality prediction on mobile cloud gaming content. *IEEE Signal Processing Letters*, 30, 324–328. doi:10.1109/LSP.2023.3255011.

[6] Mohamed Hegazy, Khaled Diab, Mehdi Saeedi, Boris Ivanovic, Ihab Amer, Yang Liu, Gabor Sines, and Mohamed Hefeeda. 2019. Content-aware video encoding for cloud gaming. In *Proc. ACM MMSys '19*, 60–73. doi:10.1145/3304109.3306222.

[7] Mahdi Hemmati, Abbas Javadtalab, Ali Asghar Nazari Shirehjini, Shervin Shirmohammadi, and Tarik Arici. 2013. Game as video: bit rate reduction through adaptive object encoding. In *Proc. ACM NOSSDAV '13*, 7–12. doi:10.1145/2460782.2460784.

[8] Chun-Ying Huang, Cheng-Hsin Hsu, Yu-Chun Chang, and Kuan-Ta Chen. 2013. Gaminganywhere: an open cloud gaming system. In *Proc. ACM MMSys '13*. doi:10.1145/2483977.2483981.

[9] Shan Jiang, Zhenhua Han, Haisheng Tan, Xinyang Jiang, Yifan Yang, Xiaoxi Zhang, Hongqiu Ni, Yuqing Yang, and Xiang-Yang Li. 2025. Real-time neural-enhancement for online cloud gaming. (2025). https://arxiv.org/abs/2501.06880 arXiv: 2501.06880.

[10] LizardByte and contributors. 2025. Sunshine: self-hosted game stream host for moonlight. (2025). https://github.com/LizardByte/Sunshine.

[11] Moonlight Game Streaming Project. 2025. Moonlight: open-source game streaming client. (2025). https://moonlight-stream.org/.

[12] Omar Mossad, Khaled Diab, Ihab Amer, and Mohamed Hefeeda. 2021. Deepgame: efficient video encoding for cloud gaming. In *Proc. ACM MM '21*, 1387–1395. doi:10.1145/3474085.3475594.

[13] OutofAi. 2025. Cudacanvas: pytorch tensor display in cuda. (2025). https://github.com/OutofAi/cudacanvas.

[14] python-mss contributors. 2025. Python-mss: cross-platform screenshots in python. (2025). https://github.com/BoboTiG/python-mss.

[15] Shakarim Soltanayev, Odysseas Zisimopoulos, Mohammad Ashraful Anam, Man Cheung Kung, Angeliki Katsenou, and Yiannis Andreopoulos. 2025. Content adaptive encoding for interactive game streaming. (2025). https://arxiv.org/abs/2511.22327 arXiv: 2511.22327.

[16] TensorFlow.js Contributors. 2025. Tensorflow.js. (2025). https://github.com/tensorflow/tfjs.

[17] TensorFlow.js Team. 2022. Tensorflow.js: platform and environment. (2022). https://www.tensorflow.org/js/guide/platform_environment.

[18] Deniz Ugur, Ihab Amer, and Mohamed Hefeeda. 2025. Decoupling video upscaling from rendering for cloud gaming. In *Proc. ACM MMSys '25*, 68–78. doi:10.1145/3712676.3714439.

[19] Unity Technologies. 2025. Unity render streaming documentation. (2025). https://docs.unity3d.com/Packages/com.unity.renderstreaming@latest/.

[20] Unity Technologies. 2025. Unity render streaming: web app docs. (2025). https://docs.unity3d.com/Packages/com.unity.renderstreaming@3.1/manual/webapp.html.

[21] Unity Technologies. 2025. Unityrenderstreaming (github repository). (2025). https://github.com/Unity-Technologies/UnityRenderStreaming.

[22] Markus Utke, Saman Zadtootaghaj, Steven Schmidt, Sebastian Bosse, and Sebastian Möller. 2022. NDNetGaming - development of a no-reference deep CNN for gaming video quality prediction. *Multimedia Tools and Applications*, 81, 3, 3181–3203. doi:10.1007/s11042-020-09144-6.

[23] Corentin Wallez and Brandon Jones. 2023. Webgpu: unlocking modern gpu access in the browser. (2023). https://developer.chrome.com/blog/webgpu-io2023.

[24] Saman Zadtootaghaj, Steven Schmidt, Saeed Shafiee Sabet, Sebastian Möller, and Carsten Griwodz. 2020. Quality estimation models for gaming video streaming services using perceptual video quality dimensions. In *Proc. ACM MMSys '20*, 213–224. doi:10.1145/3339825.3391872.

[25] Eduard Zamfir, Marcos V. Conde, and Radu Timofte. 2023. RT4KSR: real-time 4k image super-resolution baseline (ntire real-time 4k sr challenge). In *Proc. CVPR Workshops*. NTIRE Real-Time 4K SR Challenge baseline.