Capsule: Efficient Player Isolation for Datacenters

Zhouheng Du, Nima Davari, Li Li, Wei Sen Loi, Nodir Kodirov
Huawei Technologies Canada
{simon.du1,nima.davari,leo.lili,wei.sen.loi,nodir.kodirov}@huawei.com

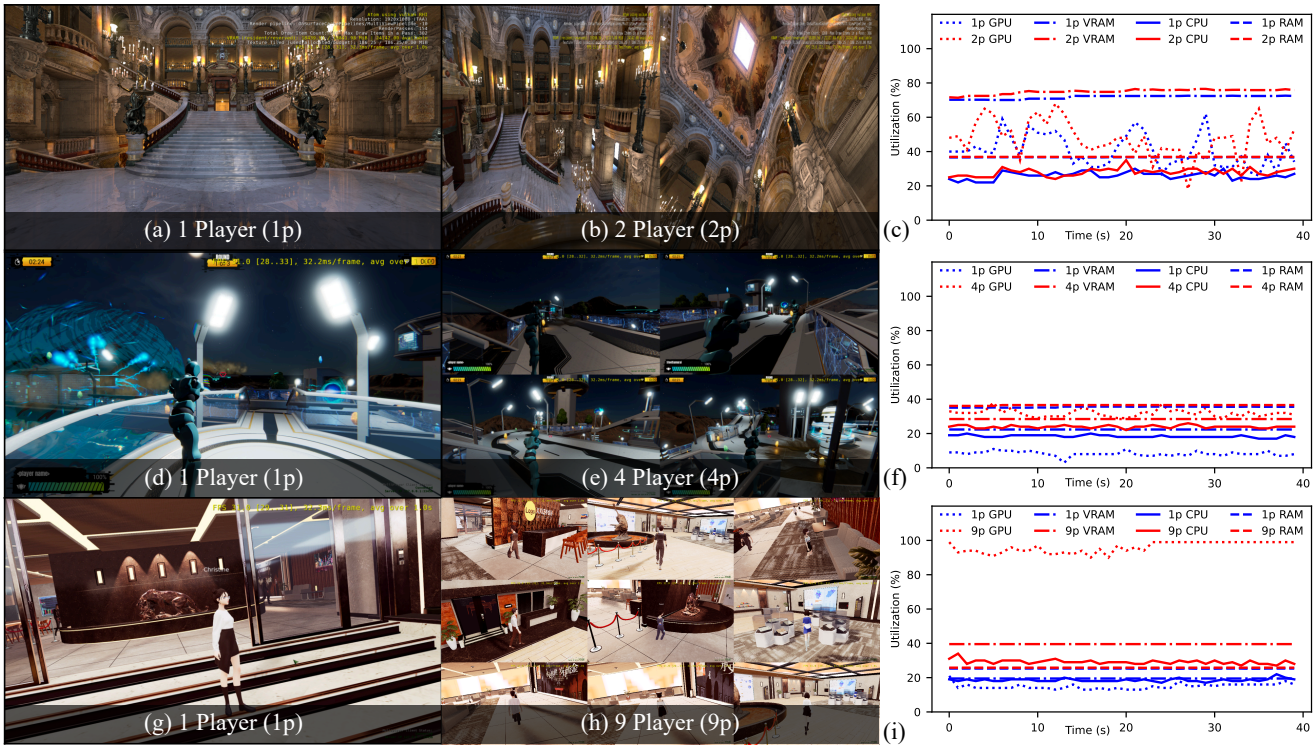

Figure 1: Three examples demonstrate applicability of Capsule to wide range of graphics intensive applications: 1) *high-graphics*: Paris Opera House to experience a digital twin of Paris Opera House (top); 2) *medium-graphics*: O3DE Multiplayer Sample, a shooting game (middle); 3) *low-graphics*: Exhibition to experience digital twin of a museum (bottom). Capsule achieves high datacenter resource utilization–GPU, VRAM, CPU, RAM–while providing lightweight and efficient player-isolation.

Abstract

Cloud gaming is increasingly popular. A challenge for cloud provider is to keep datacenter utilization high: a non-trivial task due to application variety. These applications come in different shapes and sizes. So do cloud datacenter resources, e.g., CPUs, GPUs, NPUs. Part of the challenge stems from game engines being predominantly designed to run only one player. For example, one player in a lightweight game might utilize only a fraction of the cloud server GPU. The remaining GPU capacity will be left underutilized, an undesired outcome for the cloud provider.

We introduce Capsule, a mechanism to seamlessly share one GPU, and other cloud servers resources, across multiple players. Sharing makes the cost of multiple players sublinear. We implemented Capsule in O3DE, a popular open source game engine. Our evaluations show that Capsule increases datacenter resource utilization by accommodating up to 2.25x more players, without degrading player gaming experience. This is the product of Capsule using up to 1.43x less GPU, 3.11x less VRAM, 3.7x less CPU, and 3.87x less RAM compared to the baseline. Capsule is also application agnostic. We ran four applications on Capsule-based O3DE with no application changes. Our experiences with four applications, three servers with different hardware specifications, including the one with four GPUs, and multi-server cluster show that Capsule design can be adopted by other game engines to increase datacenter utilization across cloud providers.

CCS Concepts

• **Computer systems organization** → *Cloud computing*; *Real-time system architecture*; • **Computing methodologies** → **Graphics systems and interfaces**.

1 Introduction

Cloud gaming is attractive for players as well as cloud providers. For players, it alleviates the deployment cost. They no longer have

to own the latest hardware (e.g., GPU) to play the game in high quality. Cloud already hosts the latest hardware, sometimes before they become publicly available [Petty et al. 2023]. For providers, it is about generating revenue while delivering the highest gaming quality. Higher the cloud datacenter utilization, higher the revenue.

However, it is challenging to achieve high datacenter utilization with gaming applications. Part of the challenge arises from games having diverse *shapes* and *sizes*. Shapes correspond to diverse resources games consume, such as CUDA cores, RT cores, and Tensor cores in GPUs, in addition to the host CPU and RAM. Sizes correspond to the differing amount of these resources games consume, e.g., a graphics-intensive game consumes the entire GPU while a graphics-light game consumes only a fraction of that GPU. Another challenge arises from datacenters having diverse hardware. A datacenter has servers with several generations of CPUs and GPUs [Patel et al. 2023]. For example, a server with the oldest GPU can accommodate only one player while with the latest GPU accommodates dozen players. Thus, cloud providers need a mechanism to share GPU, and other resources, across multiple players.

Games are not the only kind of cloud application that require server resource multiplexing. Other cloud workloads do too and resource virtualization has been the primary solution for over 20 years [Barham et al. 2003]. Broadly, there are three virtualization categories: Virtual Machines (VM) [AWS 2025], containers [Google 2025], and lambdas [Azure 2025], as shown in Figure 2. The container approach is well-fit for games because (1) it does not require knowing the application semantics, which makes the solution applicable to wide range of games, and (2) it does not impose high overhead, i.e., multiplexing mechanism itself consumes insignificant resources, e.g., CPU cycles. Lambdas suffer (1) and VMs suffer (2). How does container-like multiplexing solution look like for gaming applications?

We propose player-level multiplexing. A player in the graphics-heavy application will continue consuming the entire GPU. However, when a GPU has sufficient capacity to accommodate two or more players in a multiplayer game, GPU resources will be multiplexed across these players. We designed, implemented, and evaluated **Capsule: an in-game-engine player isolation mechanism for multiplayer games**. Capsule also allows cross-player *sharing*. For example, when two players enter a room and have a shared game asset in their view, we can reuse the asset geometry across these two players, without players noticing. Sharing offers cloud providers with sublinear resource usage growth for linear player increase: the phenomena we call *sublinear resource footprint*.

We implemented Capsule in O3DE, a popular open source game engine [O3DE 2025b]. Our evaluations show that Capsule-based O3DE can increase datacenter resource utilization by accommodating up to 2.25x more players, without degrading player experience. Capsule is also application agnostic. We ran four applications on Capsule with no application changes. Our experiences show that Capsule design is generalizable and can be adopted by other engines to increase datacenter utilization across cloud providers.

2 Requirements

There are four requirements for a player-isolation mechanism in cloud, in order of their importance:

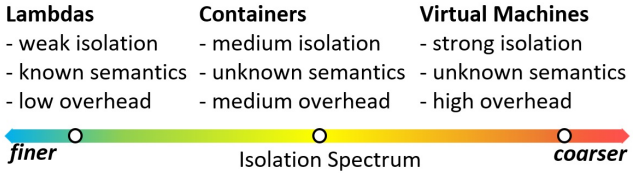

Figure 2: Existing isolation mechanisms in the cloud. Lambdas are lightweight but require knowledge of application semantics. Virtual Machines are agnostic to semantics but are heavyweight. The container approach strikes the right balance for gaming applications.

- **R1: Transparent**: Players should be unaware of other players sharing cloud resources. A player experience, such as, input latency, output streaming quality, and frames-per-second (FPS), should not degrade due to other players.
- **R2: Compatible**: Player isolation should not require significant changes to run existing applications, best if no application changes are required. The workflow for developing a new application should also remain near identical, if not exactly identical.
- **R3: Lightweight**: Isolation mechanism itself should not consume significant system resources, e.g., CPU and RAM.
- **R4: Efficient**: Maximize cross-player sharing. For example, resource (e.g., CPU) footprint of the second player should be less than that of the first player because the second player can reuse some computation results from the first player.

Capsule satisfies all four requirements. However, for example, the process-level-isolation would satisfy **R1** and **R2**, but violate **R3** and **R4**. The process-level-isolation is achieved when we simply run separate game engine process in a cloud server for each player. **R3** is violated because spawning and managing separate OS process is not as lightweight as handling all computation within the same process. **R4** is violated because players are unable to share computation results, which is the by-product of the process-level-isolation by design because the OS processes operate on a separate memory address (unless another mechanism is used for cross-process-memory-sharing). See Section 6 for further discussion of functional and performance transparency (**R1**).

3 Design and Implementation

We designed Capsule to satisfy all four aforementioned requirements. Figure 3 shows Capsule architecture, along with other essential modules in cloud deployment. Capsule is a new module in O3DE. Capsule communicates with different system components, such as audio system, input system, rendering system (includes both audio and video rendering), and game logic (event system). Capsule leverages Entity–Component–System (ECS), an existing game engine architecture [Wikipedia 2013]. ECS makes it convenient to represent game world objects. An ECS-based game engine contains *entities* that have data *components* and *systems* to operate on those components. This architecture is widely adopted by modern game engines [Unity 2025], including O3DE.

As shown in Figure 3, players connect to the cloud over the wide area network, e.g., Internet. Players' entry point is the Streaming

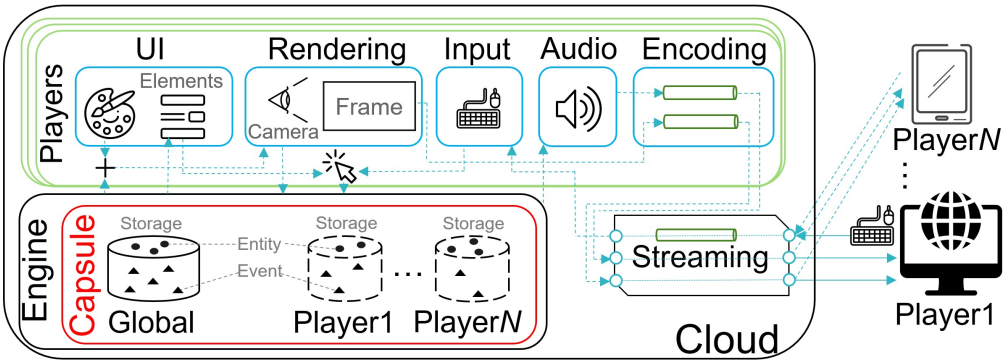

Figure 3: Capsule-based cloud architecture. Capsule provides player isolation by relying on Capsule Storage for player state management.

module, which creates a separate game session for each player, isolating their inputs, such as keyboard and mouse, and outputs, such as video streams. The Streaming module passes the player-specific input to a separate game session, which includes player-specific UI, Rendering, Input, Audio, and Encoding states. These states are reflected in the game, but are managed inside the engine. In other words, application will perceive all player-specific states to be isolated from each other at the engine level, i.e., as if each player (or in fact, each game instance) has an engine of its own. In Capsule-based O3DE, all of these players share one engine.

Capsule distinguishes different players' input, output, and behaviours by using Capsule Storage. Capsule Storage manages two constructs: entities and events. Entities are game objects, such as a car, a light, an avatar. Events, or gameplay events, include in-game events, such as running, jumping, exploding. They are predesigned by the game developer and are written in the game script. Entities and events determine the behavior of each player and of the global environment. They also determine the final rendered frame.

As shown in Figure 3, Capsule has two types of Storage: global and local. There is only one global storage in the entire engine. All players share the entities and events inside the global storage. There are one or more local Capsule Storages, one for each player. Entities inside the local storage are visible only to storage-owner-player except for the *player controlled network entities*, such as the player avatar whose state is shared with other players. For example, when a player jumps, that avatar jump should be visible to everyone. Per-player events are contained within the local storage. Capsule isolates player-specific tasks at runtime by directing player-specific entities and events to the respective player-owned local storage.

Figure 4 shows how Capsule handles per-player entity tracking using global and local storage. There are seven kinds of constructs, starting with Predefined Global Entity and ending with Player Controlled Network Entity. Not all of these constructs are exposed to the game developer. In fact, the game developers design games as if they did for vanilla (non-Capsule) O3DE, e.g., declare an entity as Network Entity (such as a museum piece in Exhibition) or Player Controlled Network Entity (such as a player avatar). Capsule manages entity division into local and global, as well as reference tracking. For example, when there are no players, there are no constructs. Optionally, Capsule could automatically create and maintain a game level cache to accelerate game loading when players join. This cache is invisible to the game.

As Figure 4 shows, players join through API calls from the Streaming module (see Figure 3). When the API call is made for the first player, predefined global entities are initialized, unless they have already been initialized via cache. During player join, all entities that the game developer specified in the initial level are spawned. For subsequent players, predefined global entities are not re-instantiated, instead, references to the original entities are created within each subsequent player's local storage. These references ensure that the player local gameplay logic remains consistent with the vanilla game-level design (for non-Capsule engine).

When the second player joins, local entities are duplicated for each player. A corresponding reference for each duplicate is maintained in the global storage. This reference mechanism ensures that player-specific gameplay logic is consistent with the vanilla design. Network entities, i.e., entities whose state and logic must be synchronized through the game server [Lu et al. 2006], are typically treated as global entities since network entities are inherently shared among players. Such treatment is consistent with legacy multiplayer architecture.

Event tracking follows analogous flow as the entity tracking. There are global events, such as time-of-the-day change in Paris Opera House, and local events, such as bullet inventory decrease after player shots in O3DE Multiplayer Sample. These distinctions also apply to the event storage, which also separates global and local scopes. Global events may trigger both global and local events, whereas local events are restricted to interactions within the same local event storage. For example, time-of-the-day change event affects all players, whereas a local inventory-update event should remain confined to that player's local storage.

A notable exception arises with player-controlled network entities, e.g., the entity that is labelled with "Dynamically Added from Game Server" in Figure 4. This entity is globally visible across all players, thereby functioning as a global entity, but its event propagation is restricted to local event storage. For instance, a player pressing the "attack" command generates its own animations and state changes without broadcasting to other players. However, when

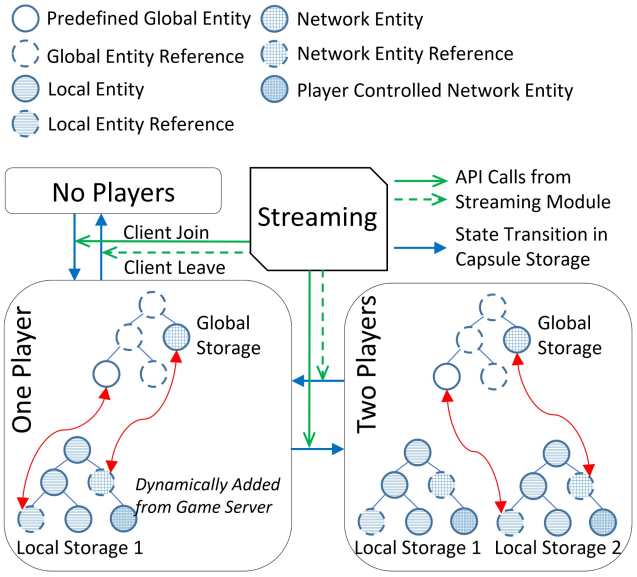

Figure 4: Player entity tracking using global and local Capsule storage. Global storage is instantiated only once. Local storage is created as players join and is destroyed when they leave the game.

this action results in interactions with other players (e.g., a collision), the corresponding events are escalated to the game server and synchronized globally, making these events visible to other players. This local-propagation-until-interaction approach ensures that player inputs remain isolated, while cross-player interactions get broadcasted across the entire shared game world.

We designed Capsule Storage (Figure 4) and Capsule architecture (Figure 3) with generality in mind. Most of the Capsule changes lies on either the ECS layer or the input and output systems of the game engine. This design maintains compatibility with other sub-systems, such as the rendering and physics systems. Therefore, it is possible, and in fact, it is encouraged, to replace these systems with multi-player-aware alternatives, without modifying Capsule. For example, a related work, On Surface Caches (OSC) [Weinrauch et al. 2023], caches and reuses the computed color values of the same world position across multiple players. The default O3DE rendering sub-system can be replaced with OSC-like alternative to further improve cross-player-sharing, without changing Capsule.

We ported four applications to Capsule-based O3DE to validate our design. All four applications nicely fit within our entity and event tracking systems. Figure 1 shows three of them for brevity. Our experience with these four applications show that Capsule-based O3DE is fully compatible (**R2**) with non-Capsule O3DE. No changes were required to these applications. Moreover, the workflow for further development of these applications on Capsule-based O3DE was identical to that of non-Capsule O3DE.

Capsule design principles can be easily adopted in other game engines thanks to ECS architecture. We also believe that four design requirements (**R1**–**R4**) are the common goals in all cloud deployments. Thus, our design and implementation decisions are applicable to other game engines and other cloud environments.

4 Methodology

We evaluated Capsule on diverse datacenter hardware. We used three different workstations: with one GPU, with two GPUs, and with four GPUs, as described in Table 1. All workstations run Windows OS to faithfully reproduce our production environment. Capsule implementation exists for Linux OS but is not as thoroughly performance evaluated as Windows. We believe results we report here are applicable to Linux environment as well. We fixed the application FPS to 30, a common minimum threshold. We read the system-wide utilization levels of GPU, VRAM, CPU, and RAM resources every second. The value on each second is the average utilization during that one-second interval; consistent with Windows performance counters [Microsoft 2025].

We compare the cloud server resource consumption of Capsule against the Baseline. For Baseline, we implemented process-level-isolation, i.e., a separate game engine process for each player. (We were unable to use production-level alternative virtualization techniques, such as NVIDIA RTX Virtual Workstation (vWS) [NVIDIA 2025b], for the baseline due to NVIDIA licensing restrictions [NVIDIA 2025c].) In Baseline, we launched the Exhibition game server [Lu et al. 2006] and then launched game clients one by one, measuring the server utilizations as players get added. The client-side evaluation is identical between Capsule and Baseline, but on the server side, after the first player, we keep adding players to the same client process (running in the cloud server), rather than creating a separate process per player (in the cloud server). This is consistent with the Capsule design in Figure 3.

We evaluated Capsule's applicability to diverse datacenter workloads by running three different applications–Paris Opera House, O3DE Multiplayer Sample [O3DE 2025a], Exhibition–on SingleGPU workstation (see Table 1). We monitored resource utilization for 40 seconds of gameplay time.

In hardware diversity experiments, we ran Exhibition on three workstations from Table 1. In these experiments, we compare resource utilizations of Capsule and Baseline with two or more players. In each experiment, the gameplay of each player is *unique*, and is *deterministic* across different hardware. It is unique because the player walking trajectories differ within the same experiment. We record a stochastic trajectory for each player in a separate experiment and replay that trajectory in Capsule vs. Baseline evaluations. For example, if there are 4 players in the experiment, each player trajectory is stochastic. Stochasity makes our evaluations unbiased, i.e., free from the selective benchmarking crime [Heiser 2025]. The view the player gets in a frame influences the rendering load for that frame, which in turn influences how much computation can be shared across multiple players in that frame. In the biased case, we would have all players have the same view, or largely overlapping view, which would unfairly make Capsule outshine the Baseline because the amount of cross-player sharing is maximized. By adopting stochasity, our evaluations are free from such bias. In fact, stochastic trajectories might undersell the Capsule benefits because players might inadvertently have less view overlap, hence less sharing, than they would otherwise have in the realistic, production

Table 1: Workstations used for Capsule evaluation. All workstations use NVIDIA GPUs. RAM and VRAM values are in GB.

Workstation	CPU	CPU Cores	RAM	GPU	VRAM	GPU Count
SingleGPU	AMD Ryzen 7 5800X	8	32	GeForce RTX 4090	24	1
DualGPU	13th Gen Intel Core i7-13700K	16	64	GeForce RTX 3090	24	2
QuadGPU	AMD Ryzen Threadripper PRO 5975WX	32	256	Fortis[1]	24+	4

deployment. We would rather undersell than be biased. Thus, the Capsule benefits we report in our experiments are conservative.

The determinism across different hardware allows our experiments stay true to our claim: hardware is different, everything else is the same. When we evaluate different hardware, we replay the prerecorded trajectories for the first player, the second player, and so on. Thus, the player generated rendering load is identical across all hardware configurations while the number of players accommodated might increase (or decrease) depending on the compute capacity of the hardware under evaluation. This exactly is the purpose of the hardware diversity experiments: evaluate if the Capsule benefits are consistent across different datacenter hardware.

In general, we chose the strongest viable baseline and compared it to Capsule. We evaluated on wide range of applications and diverse datacenter hardware while faithfully replicating our production environment and remaining bias free.

5 Evaluation

We evaluate how efficiently Capsule accommodates multiple players. We run three applications on three datacenter servers. We first evaluate all three applications on SingleGPU workstation, with the strongest GPU. We then evaluate Exhibition application on three workstations, and on a multi-server cluster.

Figure 1 shows three applications on SingleGPU workstation. Figure 1(a) and Figure 1(b) show game server view of Paris Opera House with single player (1p) and two players (2p), respectively. Figure 1(c) shows resource utilizations for these two environments. Unlike other applications used in our experiments, GPU usage in Paris Opera House has high variance. For example, in some small intervals, 2p utilization line is below 1p utilization. This is partially due to the utilization capturing noise, but is mostly due to view angle of the players. For example, between 25th-28th seconds, if two players in 2p environment happen to view the corner of the wall, while the single player in 1p environment views the distant place with many polygons, 1p GPU utilization will be higher. However, if we summarize the GPU utilizations across the entire 40-second gameplay, without focusing on a small time intervals, the average 2p GPU utilization (≈50%) is ≈10% higher than 1p (≈40%). This is expected: with Capsule, the second player imposes a sublinear ≈10% GPU cost.

This delta is smaller, but is more **significant** in other resources (VRAM, CPU, RAM). If the delta is the smallest, i.e., zero, 1p and 2p lines overlap. This means that 2p utilization of that resource is identical to that of 1p, which means the second player came for free (for that resource). In Figure 1(c), this is almost the case for VRAM, CPU, and RAM: the second player has a small extra cost. Thus, the cost of additional players is sublinear for these resources.

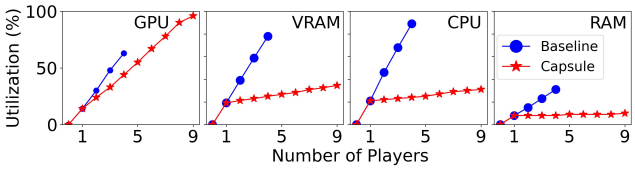

Figure 5: Scalability of Capsule as we increase number of players. Capsule and Baseline host up to 9 and 4 players, respectively, on the Exhibition application. Capsule accommodates up to 2.25x more players thanks to sublinear resources increase per added player.

Capsule is unable to accommodate more than two players in Paris Opera House application. The FPS drops below the threshold (30) with the third player due to CPU bottleneck. This is not visible in Figure 1(c), i.e., CPU utilization is only ≈30% for 1p as well as 2p. This is misleading because CPU utilizations are reported and are plotted for all 8 cores (see SingleGPU workstation in Table 1) while there is only one main thread for all players, which runs on a single CPU core. That thread is the bottleneck. We can improve Capsule performance by spreading players across CPU cores. We have not done so, yet, because in most games, GPU is the bottleneck (as evident in our third application, Exhibition). However, extending Capsule to use different CPU cores for different players is the right future work to make Capsule applicable to diverse applications.

The sublinear cost conclusion holds in O3DE Multiplayer Sample [O3DE 2025a] and Exhibition applications, in Figure 1(d)–(f) and Figure 1(g)–(i), respectively. For example, as shown in Figure 1(i) for Exhibition application, one player consumes ≈18% of GPU while 9 players consume around ≈99%: a sublinear per-player increase. Similarly, CPU consumption increases sublinearly: only by ≈10% point with 9 players (≈29%) vs. one player (≈19%). Note that O3DE Multiplayer Sample also suffers aforementioned single-thread bottleneck. Exhibition does not suffer it and therefore can achieve over 99% GPU utilization for 9 players. These results demonstrate that Capsule, as it is implemented now, brings greater benefit to graphics-heavy application, i.e., when GPU is bottleneck. Our future work will alleviate CPU bottleneck.

Figure 5 shows average resource consumption increase as players join Exhibition application in Figure 1(h). Note that we adjusted for the effect of game processes only, by subtracting base utilizations from the measured values. Capsule supported up to 9 players without dropping FPS below 30, while baseline sharply dropped to single-digit FPS after 4 players due to CPU contention. At the peak of the Baseline (4 players), Capsule used **1.43x less GPU**, **3.11x**

[1]Exact GPU model is not disclosed. VRAM "24+" means it has more VRAM than the above two. Fortis is our custom label, means *strong* in Latin.

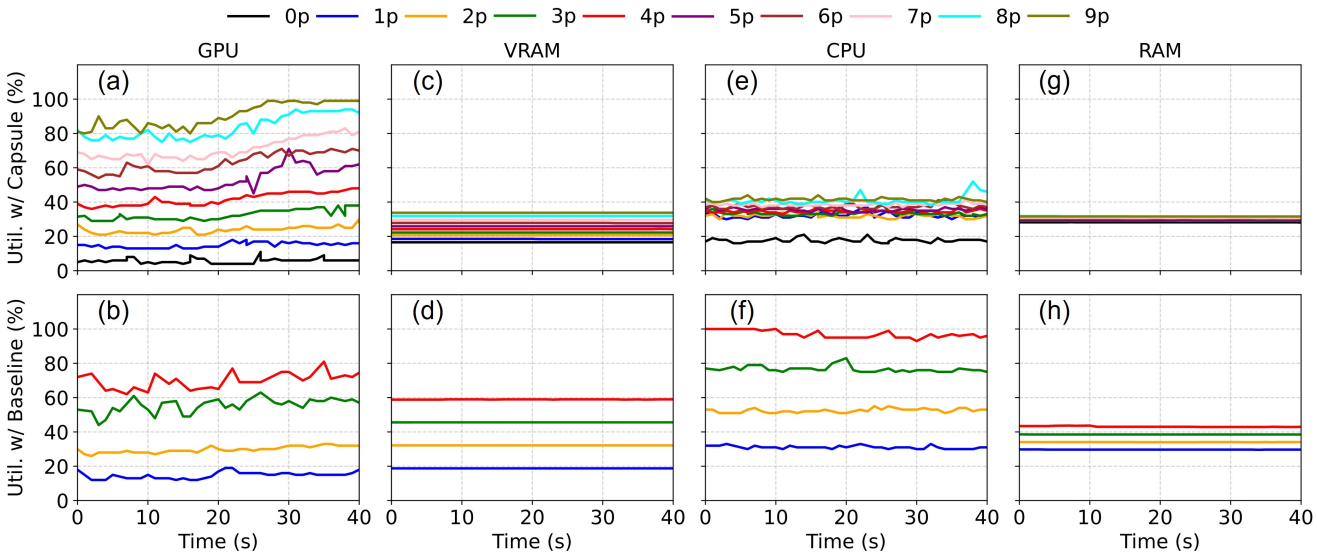

Figure 6: System-wide utilizations with Capsule and Baseline on SingleGPU workstation.

less VRAM, **3.7x less CPU**, and **3.87x less RAM** compared to the Baseline. Thanks to these savings, Capsule accommodates more players, beyond the Baseline. As expected, the Baseline resource consumption increases linearly with the number of players. However, the increase in Capsule is sublinear thanks to cross-player sharing, i.e., Capsule requires a sublinear amount of extra VRAM, CPU, RAM past the first player. This trend also holds in GPU consumption, but since each player has a different view of the scene, sharing of GPU computation is lower than in other resources. The GPU utilization benefit of Capsule could be further improved if the application uses more shareable rendering techniques, such as shadowmaps, global illumination, and cross-view diffuse and effects sharing [Weinrauch et al. 2023].

Figure 6 gives a finer view of the Figure 5 experiment by showing per-second resource utilization during 40 second gameplay, for different number of players. 0P represents the initial state where the game server and the game client are launched, but no players are created yet. Results from this diversity evaluation are consistent with our earlier per-player resource footprint experiment, i.e., Capsule accommodates up to 9 players while Baseline becomes CPU bottleneck after 4 players. Thus, 2.25x more players with Capsule.

These 2.25x savings are thanks for Capsule's ability to multiplex server resources across multiple players. Figure 6(a)-(b) show GPU utilizations for different number of players. 0p line in Figure 6(a) shows ≈10% GPU utilization when there are zero players, i.e., only the game server, an empty game client, and OS background processes are running. With 1 player (1p), GPU utilization in Capsule (Figure 6(a)) is similar to that of Baseline (Figure 6(b)): both are ≈20%. However, as Figure 6(b) shows, with the Baseline, the GPU utilization reaches up to 37% with two players, up to 62% with three players, and up to 81% with four players. Thus, each player imposes linear, ≈20% GPU overhead. On the other hand, as Figure 6(a) shows, with Capsule, the overhead is sublinear: up to 30% with two players, up to 40% with three players, up to 50% with four players, and so on until up to 100% with nine players.

The reason we stop with nine players (9p) in Figure 6(a) is because GPU becomes bottleneck after nine players, which causes the game FPS to fall below the acceptable threshold (30 FPS). However, the reason Baseline stops after four players (4p) is the CPU bottleneck, not the GPU. As Figure 6(f) shows, the CPU becomes a bottleneck with the fourth player, game dropping below 30 FPS. On the other hand, as Figure 6(e) shows, Capsule is able to multiplex CPU resources across multiple players, even better than multiplexing GPU resources. This is because Capsule is able to achieve higher cross-player-sharing for CPU computation than for GPU computation, e.g., Capsule consumes ≈20% CPU with zero players, ≈32% with one player, ≈33% with two players, and so forth until only ≈40% (up to 50%, briefly) with nine players (9p).

Similar trend holds for VRAM (Figure 6(c)-(d)) and RAM resources (Figure 6(g)-(h)). Figure 6(c) shows sublinear VRAM footprint with Capsule while it is clearly linear with Baseline in Figure 6(d). Figure 6(g) shows sublinear RAM footprint, even more sublinear than in the VRAM, while the Baseline imposes linear per-player RAM resource footprint.

We also evaluated Capsule and Baseline with two GPUs on DualGPU workstation. For Baseline, we created three processes that use GPU1 and another three processes that use GPU2. For Capsule, we use a single process, but in the game engine that runs in that process four players get assigned to GPU1 and the other four get assigned to GPU2. Thus, there are eight players in DualGPU.

Figure 7 shows the results on DualGPU workstation where Baseline hits the GPU bottleneck (unlike in Figure 6). However, this time, Capsule brings only 33% benefit, i.e., it supports up to four players while Baseline supports at most three (with 30 FPS). Capsule is more effective on SingleGPU workstation than on DualGPU workstation because the former's GPU is significantly (>60%) stronger: RTX 4090 with 16,384 CUDA cores of 2.2 GHz clock frequency on the

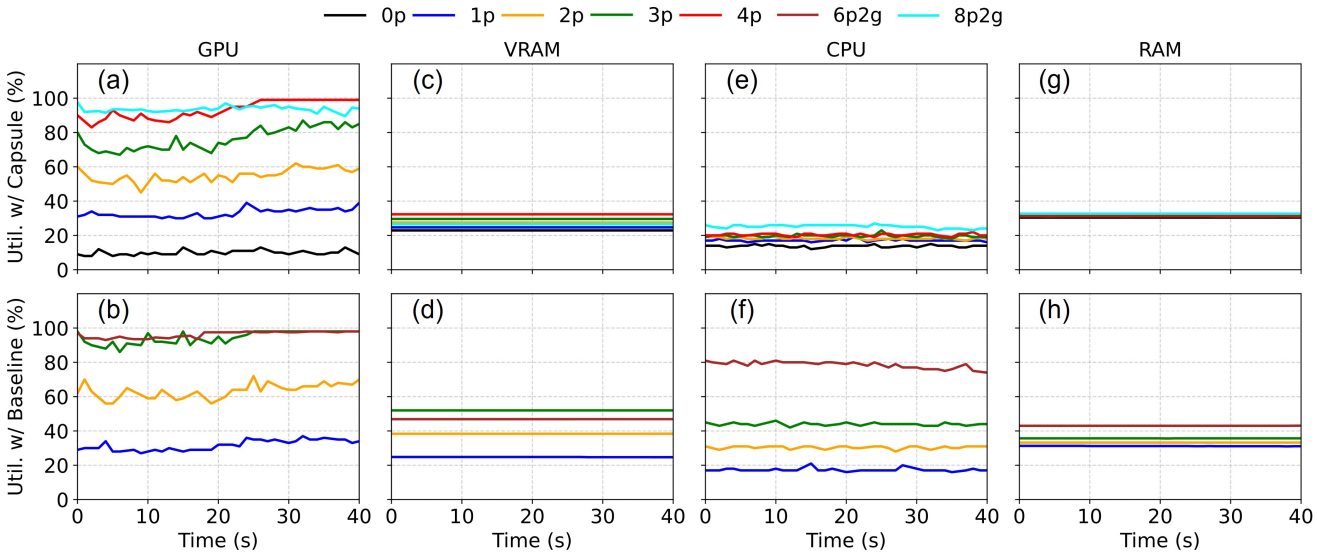

Figure 7: System-wide utilizations with Capsule and Baseline on DualGPU workstation.

former vs. RTX 3090 with 10,496 CUDA cores of 1.4 GHz clock frequency on the latter. Capsule, primarily being a GPU multiplexing technique, has less room to shine with the weaker GPU. Thus, a GPU becomes bottleneck with four players in Capsule (three players in Baseline) on DualGPU workstation while that bottleneck is hit with nine players in Capsule (four players in Baseline) on SingleGPU workstation.

Figure 7(a) shows GPU utilizations with up to eight players on two GPUs (8p2g) with Capsule and Figure 7(b) shows up to six players (6p2g) with Baseline. Note that GPU and VRAM utilizations in Figure 7 are the average of both GPUs at any given time. We capture system-wide utilizations, which include background OS processes. Thus, in these experiments, GPU1 has higher GPU and VRAM usage compared to GPU2 due to background processes running on GPU1. Therefore, when we take averages across two GPUs, per-player averages become lower than that of the players' in single GPU experiments. For example, in Figure 7(d), 3p VRAM utilization with Baseline is ≈53% while it is ≈48% in 6p (from the average of two GPUs). This means 53-48≈5% VRAM utilization was due to background processes. Similar background overhead applies to all other multi-GPU experiments.

Figure 7 findings are consistent with the ones from Figure 6. Figure 7(a) shows sublinear multiplayer resource footprint for GPU resource, i.e., it starts with ≈37% utilization with one player (1p), reaching ≈100% on single GPU with four players (4p), and reaching ≈100% on two GPUs with eight players (8p2g). On the other hand, with Baseline (Figure 7(b)), GPU utilization is ≈37% with one player (1p), reaching ≈100% with three players on single GPU (3p), and reaching ≈100% with six players on two GPUs (6p2g). This sublinearity is even more evident for VRAM resource (Figure 7(c)-(d)), i.e., it grows only ≈9% points with Capsule (≈22% in 1p vs. ≈31% in 4p in Figure 7(c)) vs. ≈28% points with Baseline (≈25% in 1p vs. ≈53% in 3p in Figure 7(d)).

As Figure 7(e)-(h) show, additional players also have sublinear CPU and RAM footprints with Capsule, as expected. For example, Baseline uses ≈**60%** point additional CPU cycles (≈20% in 1p vs. ≈80% in 6p2g in Figure 7(f)) and ≈**10%** point additional RAM to go from one player to six players (≈32% in 1p vs. ≈42% in 6p2g in Figure 7(h)). However, Capsule needs less than **11%** (vs. Baseline≈60%) point additional CPU cycles (≈18% in 1p vs. ≈29% in 8p2g in Figure 7(e)) and ≈**4%** (vs. Baseline≈10%) point additional RAM (32% in 1p vs. 35% in 8p2g in Figure 7(g)) to go from 1 player to 8 players.

Figure 8 shows the results on QuadGPU workstation.[2] The number of players hosted in Baseline and Capsule is similar as in the DualGPU workstation, as workstations have comparable GPUs, i.e., RTX 3090 and Fortis yield comparable performance in Exhibition application. QuadGPU workstation has four GPUs and has a very powerful CPU. We allocate players across multiple GPUs. Each GPU can accommodate up to four players with Capsule and only three players with Baseline (with 30 FPS). Thus, as shown in Figure 8(a), Capsule can host up to 16 players on four GPUs (16p4g) before the system becomes GPU bottleneck.

As Figure 8(a)-(b) show, additional players consume sublinear GPU resource with Capsule. For example, Baseline uses ≈**65%** point additional GPU (≈30% in 1p vs. ≈95% in 6p2g in Figure 8(b)) to go from one player to six players, while Capsule needs ≈**67%** (vs.

[2]With Baseline, we should have been able to host up to 12 players on four GPUs, but we show only up to six players in Figure 8. We actually were able host eight players. GPU driver crashes with an unexpected error when launching the ninth players. We exclude the third, partially utilized GPU, from Figure 8 to avoid inconsistency in averaging. As explained earlier, we report average GPU utilizations for our multi-GPU workstations. This assumes that all GPUs host equal number of players. This is not the case when the third GPU hosts only two players (before GPU crash) while there are three players on first and three players on second GPUs. Therefore, in Figure 8, we exclude (two players on) GPU3 from our calculations and just average the utilizations for six players (6p) on GPU1 and GPU2. We believe the conclusions we draw are still valid after GPU3 exclusion because our conclusions rely on cross-GPU averages. Even if GPU3 (and further GPU4) did not crash, and we would still get three players on those GPUs, Baseline accommodating one less player per GPU than Capsule, just like in GPU1 and GPU2. Thus, our conclusions would still hold.

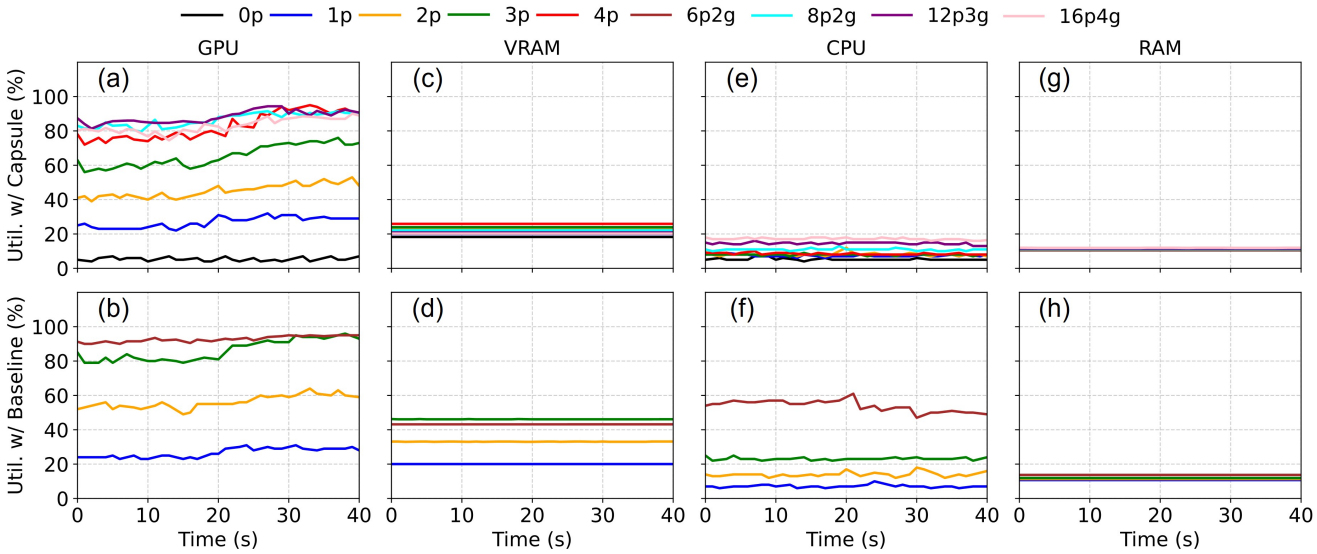

Figure 8: System-wide utilizations with Capsule and Baseline on QuadGPU workstation.

Baseline≈65%) point additional GPU (≈30% in 1p vs. ≈97% in 16p4g in Figure 8(a)) to go from one player to 16 players. Similar sublinearity holds in VRAM, CPU, and RAM resources:

- VRAM: Baseline uses ≈**26%** point additional VRAM (≈20% in 1p vs. ≈46% in 6p2g in Figure 8(d)) to go from one player to six players, while Capsule needs ≈**2%** (vs. Baseline≈26%) point additional VRAM (≈20% in 1p vs. ≈22% in 16p4g in Figure 8(c)) to go from one player to 16 players.
- CPU: Baseline uses ≈**51%** point additional CPU (≈8% in 1p vs. ≈59% in 6p2g in Figure 8(f)) to go from one player to six players, while Capsule needs ≈**11%** (vs. Baseline≈51%) point additional CPU (≈8% in 1p vs. ≈19% in 16p4g in Figure 8(e)) to go from one player to 16 players.
- RAM: Baseline uses ≈**6%** point additional RAM (≈10% in 1p vs. ≈16% in 6p2g in Figure 8(h)) to go from one player to six players, while Capsule needs ≈**2%** (vs. Baseline≈6%) point additional RAM (≈10% in 1p vs. ≈12% in 16p4g in Figure 8(g)) to go from one player to 16 players.

We also evaluated Capsule in multi-server setup by connecting DualGPU and QuadGPU in a cluster. We were able to replicate our results from Figure 7 and Figure 8, i.e., 8 players on the former workstation and 16 players on the latter one. All 24 players were connected to the same game server and they had smooth gameplay with 30 FPS, as expected. Thus, we believe Capsule's sublinearity benefits will remain applicable to datacenters of larger scales.

6 Discussion

Capsule has some limitations, which can be further improved. For example, transparency requirement (**R1**) has two aspects: *functional* and *performance*. Functional aspect refers to the gameplay experience, e.g., if two players are in the same game session and are colocated on the same engine and the same GPU, one player jumping should not interfere with the other player's jump. This is the essential part of the **R1**. Capsule and process-level-isolation, which we used as the baseline, satisfy this requirement. There is also performance aspect, e.g., one player exhaustively using GPU resources by frequent jumps should not reduce FPS for the second player. This is often called noisy-neighbour problem in cloud [Novaković et al. 2013]. Process-level-isolation does not satisfy this requirement. Neither does Capsule, at least as implemented now.

Performance transparency, or performance isolation, in the current Capsule implementation is achieved in an indirect way: player rate-limiting after worst-case-analysis. When the game is deployed in cloud, the game goes through offline performance profiling. The profiler outputs the FPS range this specific GPU sustained during an exhaustive gameplay, similar to code-coverage-based analysis in software engineering [Grechanik et al. 2012]. We then do the worst-case-analysis, i.e., derive the maximum number of players this GPU can host in the worst case, which is when all players perform graphics-heavy operations, concurrently. For example, if profiler outputs FPS=[130-150] range after the offline analysis, and the game developer requires at least 30 FPS for smooth gameplay, we deploy at most 130/30≈4 players on this GPU. We repeat offline profiling and analysis for each GPU flavor in the cloud. Some GPU flavors might get disqualified altogether for this application if they cannot offer the minimum developer-required FPS even for 1 player.

Although indirectly achieving performance transparency works, in the future, we would like to explore way of achieving it directly. In this exploration, our guiding principle would be maintaining lightweightness (**R3**) because unless done with care, stronger performance isolation mechanisms add non-trivial overhead, as commonly known as the *virtualization tax* [Keller et al. 2010].

Capsule also introduces player fate-sharing, which might degrade player fault-tolerance. For example, when a server GPU fails, all players running on the game engine hosted on that GPU also fail. Process-level-isolation also suffers this limitation. Fate-sharing

is not about a player causing a GPU failure (or any system component failure), but is about hardware failures themselves. Without Capsule, there is only one player per game engine, one engine per GPU, and the GPU failure impacts only that single player. However, fate-sharing is an inherent problem, i.e., the moment we decide to multiplex GPU resources across players we accept fate-sharing; just like students share a bus ride everyday: a broken bus delays everyone's school arrival. Fate-sharing is also common and inevitable in cloud: VMs, containers, lambdas all suffer it (Figure 2), e.g., when a server fails, all VMs hosted on that server fail. The disadvantages of the fate-sharing can be ameliorated by other mechanisms, such as player migration with player state replication, as it has been done for VMs over 20 years ago [Clark et al. 2005].

7 Related Work

The concept of accommodating multiple players on the same game engine is already well established under the term *Local Multiplayer* [Karhulahti and Grabarczyk 2021]. Before online multiplayer became prevalent, multiplayer games were mostly played offline on the same console where the screen would be divided between the players [Nintendo EAD 1992; Rare 1997]. Each player would either have their own controller (connected to the same console) or share a subset of keys on the same keyboard. Exclusive audio or mouse control was seldom an option. While local multiplayer games are still popular [Ghost Town Games and Team17 Digital Ltd. 2020; Hazelight Studios 2021; Larian Studios 2023] and have support in many modern game engines [Epic Games sent; Godot Engine community and Godot Foundation sent; Unity Technologies sent], the majority of games only support multiplayer over the network, which we call *network multiplayer games*. Network multiplayer approach relaxes input devices sharing, display, and other restrictions, such as supporting larger number of players. However, a common limitation of all network multiplayer games is the inability to reuse compute and memory across players, which was possible with on-the-same-console approach.

With cloud gaming becoming viable in the last decade, many platforms now provide remote servers to host games on the cloud [Microsoft 2019; NVIDIA 2015; Sony Interactive Entertainment 2022]. To share powerful cloud server resources between multiple players, cloud providers might virtualize and partition the hardware, e.g., GPU and CPU, granting only a slice of these (virtual) resources to each player [NVIDIA 2020, 2025]. The cross-player isolation is often provided by running each game session on a separate virtual machine (VM) [NVIDIA 2025a,b]. However, just like pre-cloud networked multiplayer games, the VM-based approaches lack cross-player compute and memory reuse ability.

Cross-player sharing concept is also used in the game servers [Lu et al. 2006]. A game server manages many players at the same time, but that server mostly coordinates cross-player state, e.g., synchronize actions, inventories, and progress. Clients still render frames by themselves, owning their hardware or using remote hardware. On the other hand, Capsule strives to achieve the efficiency (**R4**) of the on-the-same-console multiplayer games in the cloud environment, without imposing screen size, input, and other restrictions. In fact, Capsule does so without compromising transparency (**R1**) and compatibility **R2**, using a lightweight (**R3**) isolation mechanism.

8 Conclusion

We designed, implemented, and evaluated Capsule: an efficient in-game-engine player isolation mechanism. Our implementation in a popular open source game engine, O3DE, shows that Capsule is application agnostic. We ported four existing applications to Capsule-based O3DE without application changes. Our experiments with these applications, three servers with different hardware specifications, including multi-GPU servers and multi-server cluster, show that Capsule can increase datacenter resource–GPU, VRAM, CPU, RAM–utilizations by accommodating up to 2.25x more players. This is the product of Capsule using up to 1.43x less GPU, 3.11x less VRAM, 3.7x less CPU, and 3.87x less RAM compared to the baseline. Capsule design can be adopted by other game engines to increase datacenter utilization across cloud providers.

Acknowledgments

Thanks to Xiaofeng Zhang, Wenxiao Zhang, Steven Yuan, Yin Wei, and people at Huawei Cloud for their support.

References

AWS. 2025. *Amazon Elastic Compute Cloud.* Retrieved September 17, 2025 from https://aws.amazon.com/ec2

Azure. 2025. *Azure Functions.* Retrieved September 17, 2025 from https://azure.microsoft.com/en-us/products/functions

Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.* 37, 5 (Oct. 2003), 164–177. https://doi.org/10.1145/1165389.945462

Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. 2005. Live migration of virtual machines. In *Symposium on Networked Systems Design & Implementation (NSDI'05)*. USENIX Association, USA, 273–286.

Epic Games. 1998–present. *Unreal Engine.* https://www.unrealengine.com/ Game development engine with native support for multiplayer, including tools for local split-screen in 3D environments, often implemented using C++ or Blueprints visual scripting..

Ghost Town Games and Team17 Digital Ltd. 2020. Overcooked! All You Can Eat. Video Game. Initial release for PlayStation 5 and Xbox Series X/S on November 10-12, 2020. A compilation featuring the remastered contents of the first two games and all DLC..

Godot Engine community and Godot Foundation. 2014–present. *Godot Engine.* https://godotengine.org/ Free and open-source game engine supporting local multiplayer with configurable split-screen modes, utilizing its built-in input and viewport management systems..

Google. 2025. *Containers at Google Cloud.* Retrieved September 17, 2025 from https://cloud.google.com/containers

Mark Grechanik, Chen Fu, and Qing Xie. 2012. Automatically finding performance problems with feedback-directed learning software testing. In *2012 34th International Conference on Software Engineering (ICSE)*. 156–166. https://doi.org/10.1109/ICSE.2012.6227197

Hazelight Studios. 2021. It Takes Two. Video Game. Platform: Multi-platform (PS4, PS5, Xbox One, Xbox Series X/S, PC, Switch); Co-op only action-adventure platformer featuring permanent local and online split-screen..

Gernot Heiser. 2025. *Systems Benchmarking Crimes.* Retrieved September 25, 2025 from https://gernot-heiser.org/benchmarking-crimes.html

Veli-Matti Karhulahti and Pawel Grabarczyk. 2021. Split-Screen: Videogame History Through Local Multiplayer Design. *Design Issues* 37, 2 (04 2021), 32–44. https://doi.org/10.1162/desi_a_00634

Eric Keller, Jakub Szefer, Jennifer Rexford, and Ruby B. Lee. 2010. NoHype: virtualized cloud infrastructure without the virtualization. In *Proceedings of the 37th Annual International Symposium on Computer Architecture* (Saint-Malo, France) *(ISCA '10)*. Association for Computing Machinery, New York, NY, USA, 350–361. https://doi.org/10.1145/1815961.1816010

Larian Studios. 2023. Baldur's Gate 3. Video Game. Platform: Multi-platform (PC, PS5, Xbox Series X/S); Features two-player local split-screen co-op for the entire main campaign, with a persistent world and narrative consequences..

Fengyun Lu, Simon Parkin, and Graham Morgan. 2006. Load balancing for massively multiplayer online games. In *Proceedings of 5th ACM SIGCOMM Workshop on Network and System Support for Games* (Singapore) *(NetGames '06)*. Association for

Computing Machinery, New York, NY, USA, 1–es. https://doi.org/10.1145/1230040.1230064

Microsoft. 2019. *Xbox Cloud Gaming*. https://www.xbox.com/en-US/play Service launched in beta in 2019.

Microsoft. 2025. *About Performance Counters*. Retrieved September 25, 2025 from https://learn.microsoft.com/en-us/windows/win32/perfctrs/about-performance-counters

Nintendo EAD. 1992. Super Mario Kart. Video Game. Platform: Super Nintendo Entertainment System (SNES); Features a two-player local multiplayer mode..

Dejan Novaković, Nedeljko Vasić, Stanko Novaković, Dejan Kostić, and Ricardo Bianchini. 2013. DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. USENIX Association, San Jose, CA, 219–230. https://dl.acm.org/doi/10.5555/2535461.2535489

NVIDIA. 2015. *GeForce NOW*. https://www.nvidia.com/en-us/geforce-now/ Service launched in 2015 as NVIDIA GRID; rebranded as GeForce NOW in 2017.

NVIDIA. 2020. *NVIDIA Multi-Instance GPU and NVIDIA Virtual Compute Server: GPU Partitioning*. Technical Report TB-10226-001_v01. NVIDIA. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/solutions/resources/documents1/Technical-Brief-Multi-Instance-GPU-NVIDIA-Virtual-Compute-Server.pdf Technical Brief.

NVIDIA. 2025a. *Frequently Asked Questions (FAQs) for GeForce NOW*. Retrieved September 25, 2025 from https://www.nvidia.com/en-us/geforce-now/faq

NVIDIA. 2025b. *NVIDIA RTX Virtual Workstation*. Retrieved September 25, 2025 from https://www.nvidia.com/en-us/design-visualization/virtual-workstation

NVIDIA. 2025c. *NVIDIA vGPU Software (RTX vWS, vPC, vApps)*. Retrieved September 25, 2025 from https://www.nvidia.com/en-us/drivers/vgpu-software-driver

NVIDIA. 2025. *NVIDIA Virtual GPU (vGPU) Technology*. https://www.nvidia.com/en-us/data-center/virtual-gpu-technology/ Accessed through the official NVIDIA Data Center documentation for the virtualization solution..

O3DE. 2025a. *MultiplayerSample Project*. Retrieved April 23, 2025 from https://github.com/o3de/o3de-multiplayersample

O3DE. 2025b. *Open 3D Engine*. Retrieved April 23, 2025 from https://github.com/o3de/o3de

Pratyush Patel, Zibo Gong, Syeda Rizvi, Esha Choukse, Pulkit Misra, Thomas Anderson, and Akshitha Sriraman. 2023. Towards Improved Power Management in Cloud GPUs. *IEEE Computer Architecture Letters* 22, 2 (2023), 141–144. https://doi.org/10.1109/LCA.2023.3278652

Harry Petty, Ivan Goldwasser, and Pradyumna Desale. 2023. *One Giant Superchip for LLMs, Recommenders, and GNNs: Introducing NVIDIA GH200 NVL32*. Retrieved April 23, 2025 from https://developer.nvidia.com/blog/one-giant-superchip-for-llms-recommenders-and-gnns-introducing-nvidia-gh200-nvl32

Rare. 1997. GoldenEye 007. Video Game. Platform: Nintendo 64 (N64); Features four-player local split-screen competitive multiplayer (deathmatch)..

Sony Interactive Entertainment. 2022. *PlayStation Plus Premium*. https://www.playstation.com/en-us/ps-plus/ Tier launched as part of the new PlayStation Plus in 2022.

Unity. 2025. *ECS for Unity*. Retrieved April 23, 2025 from https://unity.com/ecs

Unity Technologies. 2005–present. *Unity*. https://unity.com/ Game development engine supporting 2D and 3D local multiplayer and split-screen functionality via the New Input System and multiple cameras..

Alexander Weinrauch, Wolfgang Tatzgern, Pascal Stadlbauer, Alexis Crickx, Jozef Hladky, Arno Coomans, Martin Winter, Joerg H. Mueller, and Markus Steinberger. 2023. Effect-based Multi-viewer Caching for Cloud-native Rendering. *ACM Trans. Graph.* 42, 4, Article 87 (July 2023), 16 pages.

Wikipedia. 2013. *Entity Component System*. Retrieved April 23, 2025 from https://en.wikipedia.org/wiki/Entity_component_system