

Article

GPU Passthrough Across Virtualization Platforms for LLM Inference: Configuration Complexity and a Small-Model Performance Baseline

Priska Steininger ^{1,†}, Manfred Pamsl ¹, Helmut Lindner ¹, Klaus Gebeshuber ¹ and Patrick Deininger ^{1,2,*} 

¹ Institute of Computer Science and Artificial Intelligence, FH JOANNEUM—University of Applied Sciences, 8020 Graz, Austria; priska.steininger@fh-joanneum.at (P.S.); manfred.pamsl@fh-joanneum.at (M.P.); helmut.lindner@fh-joanneum.at (H.L.); klaus.gebeshuber@fh-joanneum.at (K.G.)

² Institute of Software Engineering and Artificial Intelligence, Graz University of Technology, 8010 Graz, Austria

* Correspondence: patrick.deininger@fh-joanneum.at

† These authors contributed equally to this work.

Abstract

Enterprises increasingly run GPU-bound workloads such as Large Language Model (LLM) inference inside virtualized infrastructure, yet practitioners have little systematic guidance on how the choice of virtualization platform affects GPU passthrough in configuration effort as much as performance. This study's primary contribution is a structured, paired comparison of configuration complexity and performance across six platforms (Proxmox VM and LXC, native KVM, OpenStack VM and Zun, and Podman) on a single NVIDIA RTX 4500 Ada GPU, emphasizing the under-documented container paths (OpenStack Zun and Podman with the Container Device Interface). The platforms differ most in configuration complexity, where OpenStack, especially Zun, proved the most demanding and Proxmox and LXC the most straightforward. As a confirmatory baseline, for a small-model, low-concurrency vLLM workload that does not saturate the device, exclusive passthrough yields statistically equivalent throughput, duration, and inter-token latency across all platforms, within 2% of bare metal; this equivalence is bounded to the non-saturating regime tested and is not a general claim. Running two containers concurrently (each capped at 45% of GPU memory, compute time-sliced) doubled the offered load and raised aggregate throughput by 12–14%, consistent with idle GPU capacity but not isolating a GPU-sharing benefit.

Keywords: OpenStack; Proxmox; LXC; Podman; Zun; virtualization; container; GPU passthrough; GPU sharing; CDI; KVM; LLM inference; vLLM



Academic Editor: Paolo Bellavista

Received: 26 June 2026

Revised: 22 July 2026

Accepted: 27 July 2026

Published: 29 July 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

1. Introduction

In contemporary IT infrastructures, virtualization is a key enabler of cloud computing, supporting the efficient, scalable allocation of physical resources among diverse applications and workloads [1–3]. Prior to virtualization, each application required a dedicated physical CPU or a separate server. In contrast, virtualization allows multiple applications to run in isolated Virtual Machines (VMs) or containers on a single bare-metal server, each with its own operating system, while maintaining isolation and performance guarantees [3,4]. Generally, virtualization provides a framework for partitioning a computer's resources into multiple isolated execution environments using technologies such as hardware or

software partitioning and emulation, significantly improving resource utilization [4]. In this vein, Bhowmik [1] defined virtualization as the abstraction of physical computing resources into a simulated form, facilitated through software, referred to as a layer of virtualization. This layer transforms physical computing resources into a virtual form. Simply put, virtualization refers to the logical abstraction of physical resources, decoupling direct user access from the underlying physical system. Thus, a software module known as the “hypervisor” (Virtual Machine Monitor, VMM) plays a crucial role in enabling virtualization [1,4] because it logically allocates and isolates physical resources [5].

At the same time, GPUs offer significantly higher overall computational performance than CPUs for demanding workloads, such as deploying Large Language Models (LLMs) in virtualized environments [6]. The rapid growth of AI workloads, particularly LLM inference serving, has made GPUs and other hardware accelerators a central component of modern high-performance and data-center computing, where they are increasingly provisioned through virtualized and cloud infrastructure rather than dedicated bare-metal clusters [7]. In this vein, GPUs from NVIDIA and AMD are among the most widely used accelerators, offering better performance and energy efficiency than standard multi-core CPUs [8]. Technically, GPUs are specialized microprocessors originally designed for graphics-intensive tasks, but their architecture makes them particularly well-suited for accelerating complex computations, surpassing the capabilities of general-purpose CPUs [3]. To enable virtual machines or containers to utilize GPUs, PCI passthrough is employed. This technology allows VMs to access a physical NVIDIA GPU directly, leveraging the physical GPU for computations and significantly enhancing performance [3]. Historically, implementing GPU passthrough in virtualization environments has been challenging, but recent advances have demonstrated that both VMs and containers can now reliably access physical GPUs and benefit from hardware acceleration [9]. This approach is particularly advantageous when devices cannot be shared, such as with the exclusive use of the GPU within the VMs of the present study. Regarding performance, PCI passthrough achieves nearly native execution speed [3].

1.1. Direct GPU Passthrough

Both Intel and AMD have developed hardware extensions to enable device passthrough. Intel’s implementation, known as VT-d, and AMD’s implementation, known as AMD-Vi, operate by allowing the system hypervisor to configure the processors’ Input/Output Memory Management Unit (IOMMU) [9]. Thus, in the context of PCI passthrough, IOMMUs are crucial. Their primary function is to connect an I/O bus capable of direct memory access (DMA) to the main memory. Typically integrated into the chipset, the IOMMU maps virtual device addresses to physical memory addresses [8,9]. Figure 1 illustrates how a 1:1 device passthrough to a VM is achieved. As demonstrated here, the DMA channel can now transfer data directly from the physical device to the VM’s memory space. Additionally, interrupts can be mapped directly into the VM [9].

Although GPU passthrough offers the significant advantage of fast, transparent access to the entire GPU [10], the primary limitation of PCI passthrough is its exclusivity. That is, a device (e.g., an NVIDIA GPU) assigned to a virtual machine via plain passthrough cannot be shared concurrently with other VMs without device-level virtualization (mediated passthrough (Intel’s gVirt/GVT-g [11]), SR-IOV-based vGPU [12], or hardware partitioning such as MIG [13]) nor can it be reassigned to another VM without being restarted or reconfigured. This constraint reduces the efficiency of resource consolidation [14,15].

Given the distinctions between virtualization and containerization, the mechanisms for direct GPU access differ accordingly, as detailed below:

Typically, a hypervisor resides between the VM and the physical hardware, managing the interactions between these components. However, with direct GPU passthrough, a VM can circumvent this layer and communicate directly with the physical I/O devices [16]. Consequently, a VM only needs to use the native graphics driver to directly access the physical framebuffer and the rendering functions of the GPU [10], bypassing the hypervisor's layer [16]. Thus, GPU passthrough facilitates near-native performance and minimal latency [3,8,16], which is particularly suitable for heavy compute-intensive tasks [16].

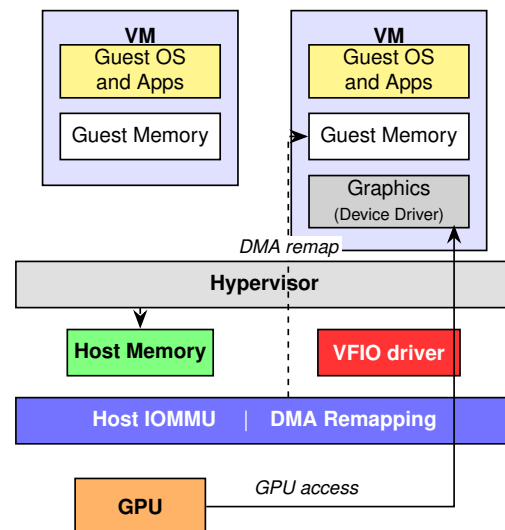


Figure 1. Direct GPU passthrough (adapted from [8–10,17]).

Accessing the GPU within a container differs from that in a VM (Figure 2). In VMs, the proprietary NVIDIA driver and its kernel modules are installed directly in the VM because the VM has its own operating system kernel. In contrast, containers require only the installation of the NVIDIA user-space libraries, such as `libcuda.so` and `libnvidia-ml.so`. However, it is necessary to install the NVIDIA driver with its kernel modules on the host system. To enable the container to access the NVIDIA devices, the appropriate cgroup device rules must be defined based on the major device numbers. Additionally, the NVIDIA user-space libraries inside the container must match the kernel driver version on the host to ensure proper GPU access [18].

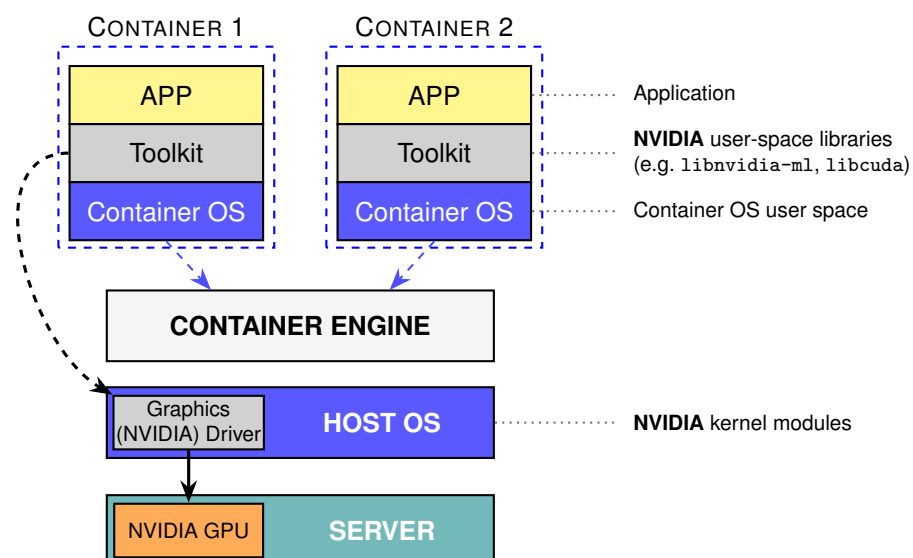


Figure 2. Container GPU access (adapted from [18]).

1.2. Virtualization Environments

Modern virtualization platforms support GPU passthrough. This study employed widely adopted and well-established platforms, including Proxmox, LXC, OpenStack, Zun, Podman and native KVM. For instance, Proxmox Virtual Environment (PVE) [19] is an open-source virtualization platform for running VMs and containers [20] which is based on QEMU/KVM and LXC [20]. Furthermore, this platform provides the ability to create and manage virtual machines, containers, high-availability clusters, storage and networks through a user-friendly web interface or a command-line interface [19,20]. Notably, Proxmox supports two virtualization technologies: First, it relies on KVM for full virtualization, which is necessary for creating VMs. Second, Proxmox utilizes LXC for lightweight container-based virtualization [20]. OpenStack is another widely used open-source software platform that provides Infrastructure as a Service (IaaS) by leveraging hypervisors such as KVM, QEMU, LXC, Xen, or VMware ESX. This software enables the deployment of private, public and hybrid clouds [10,21,22]. Nova Compute, OpenStack's compute service, supports several of these hypervisors through pluggable drivers, including the libvirt/KVM driver used in this study [23]. Beyond these virtualization platforms, the container ecosystem encompasses a range of container engines. Notably, Docker is the most widely recognized implementation, followed by other engines such as Podman, CRI-O and LXD [24]. In addition to Docker within OpenStack, this study utilizes Podman (POD Manager) as the container engine responsible for managing containers, images, mounted volumes and pods comprising container groups [24,25].

2. Related Work

GPU passthrough for virtualized environments has been studied for over a decade, primarily for high-performance computing, desktop clouds, and cloud gaming. Walters et al. [8] compared GPU passthrough performance across KVM, Xen, VMware ESXi, and LXC for CUDA and OpenCL applications, and Younge et al. [26] evaluated GPU passthrough in Xen for high-performance cloud computing; both reported near-native performance. Yang et al. [3] and Jo et al. [27] implemented GPU virtualization through PCI passthrough, while Shea and Liu [9] analyzed passthrough performance for cloud gaming and Zhou et al. [28] examined KVM-based passthrough in a desktop cloud. Closer to the present setting, Bentele et al. [10] described a GPU-accelerated virtual desktop infrastructure for OpenStack. Since Walters et al. [8] already benchmarked LXC and KVM passthrough under CUDA/OpenCL, the present study's LXC and native-KVM arms revisit those platforms under an LLM-serving workload rather than introducing them; the genuinely new platform coverage is OpenStack Zun and Podman with the Container Device Interface.

Beyond exclusive passthrough, a substantial body of work targets sharing a single GPU among multiple workloads, along a spectrum from mediated pass-through that virtualizes the device at the driver level [11], through hardware partitioning and process-level co-execution, to the software-level memory-fraction sharing with time-slicing used here. Hardware and runtime mechanisms include NVIDIA Multi-Instance GPU (MIG), which partitions a data-center GPU (Ampere generation or newer) into hardware-isolated instances [13], and the Multi-Process Service (MPS), which lets kernels from multiple processes co-execute spatially on one device rather than being time-sliced [29]. At the systems level, Salus provides fine-grained GPU-sharing primitives for deep-learning jobs [30], GSLICE controls spatial GPU sharing for scalable inference [31], and AntMan dynamically co-schedules deep-learning workloads on shared GPUs [32]. Complementary approaches virtualize or schedule the GPU at the system level: Cricket provides a transparent GPU-virtualization layer for multitenant CUDA execution [15], RPC-based GPU virtualization with partial task migration mitigates memory starvation when virtual machines share a

GPU [14], and fine-grained, balance-aware schedulers raise GPU utilization for containerized deep-learning clusters [33]. For LLM serving specifically, throughput is governed largely by memory management and batching efficiency [34,35]; more recent systems push further by disaggregating the prefill and decoding phases across GPUs [36] and by sharing a GPU spatio-temporally across serverless inference functions [37]. More recent serving systems extend this line of work: runtime request rescheduling with live migration across instances improves tail latency and utilization [38], structured-program execution with prefix sharing (RadixAttention) raises throughput for repeated and branching workloads [39], and KVCache-centric prefill/decoding disaggregation trades storage for computation to scale chatbot serving under service-level objectives [40]. These systems are built for throughput and latency under heavy, often saturating load, which is the regime that the small-model, low-concurrency baseline evaluated here does not reach; they mark the conditions under which platform-level effects on GPU inference are most likely to appear. A recurring finding across this literature is that a single model often under-utilizes a modern GPU, and systems such as GSLICE and AntMan report that controlled co-location can substantially raise aggregate throughput and device utilization [31,32]. These results provide context for the shared-GPU result reported in Section 4.2, subject to two important caveats: those systems measure GPU utilization directly whereas the present study infers it from aggregate throughput alone, and they rely on spatial co-execution (via MPS or hardware partitioning) whereas the present setup uses temporal time-slicing, so the reported gain magnitudes are not directly comparable.

Three gaps remain. First, prior comparisons focus on hypervisors (chiefly Xen, KVM, and VMware) and rarely include modern container platforms such as OpenStack Zun or Podman with the Container Device Interface (CDI) [41]. Second, existing studies rely on HPC, graphics, or synthetic CUDA/OpenCL benchmarks rather than large language model (LLM) inference serving, which is now a dominant enterprise GPU workload. Third, performance is typically reported in isolation, without a paired assessment of the configuration effort each platform demands. This study addresses these gaps by jointly evaluating six widely used platforms (Proxmox VM and LXC, native KVM, OpenStack VM and Zun, and Podman) under an LLM inference workload, and by pairing the performance measurements with a structured configuration-complexity analysis.

Accordingly, this study addresses three research questions:

- RQ1** Does the choice of virtualization platform affect GPU-accelerated LLM inference performance under exclusive passthrough?
- RQ2** How does sharing a single GPU between containers compare to exclusive allocation for this workload?
- RQ3** How do the platforms differ in the configuration complexity required to enable GPU passthrough?

This paper is an empirical benchmarking and deployment study, not a new virtualization or GPU-sharing mechanism; its contribution is evidence and structured engineering experience rather than a novel system. Concretely, it makes four contributions. First, it provides a paired evaluation that reports GPU-passthrough performance alongside a structured configuration-complexity analysis for the same set of platforms, a pairing absent from prior passthrough benchmarks that report performance in isolation. Second, it extends passthrough benchmarking from the traditionally studied hypervisors (chiefly Xen, KVM, and VMware) to two under-documented container paths, OpenStack Zun and Podman with the Container Device Interface, while the LXC and native-KVM arms revisit well-studied baselines [8] under a current software stack. Third, it uses a large language model inference-serving workload (vLLM) rather than the HPC, graphics, or synthetic CUDA/OpenCL benchmarks of prior work, matching a now-dominant enterprise GPU

use case. Fourth, it distills the results into explicit, regime-bounded platform-selection guidance (Section 5.1). The performance measurements themselves are a confirmatory baseline: near-native passthrough performance is the expected result from prior work, so the novelty lies in the paired complexity axis and the container-path coverage, not in the finding of equivalence.

3. Materials and Methods

The virtualization environment employs a dedicated bare-metal server equipped with 512 GiB of RAM, approximately 385 GiB of local disk storage, a single AMD socket and an NVIDIA Corporation AD104GL RTX 4500 Ada Generation GPU. To ensure comparability across the LLM benchmarks, each instance is configured with 32 GiB of RAM, 24 CPU cores, and 24 GiB of dedicated GPU memory, with the GPU assigned exclusively to the VMs. Table 1 provides a detailed overview of the hardware specifications.

Table 1. Hardware configuration of the virtualization environment.

Component	Description
RAM	512 GiB DDR5 (8 × 64 GiB Micron MTC40F2046S1RC48BA12, 4800 MT/s)
CPU	AMD EPYC 9454 48-Core Processor
GPU	NVIDIA Corporation AD104GL RTX 4500 Ada Generation with 24 GiB VRAM

As illustrated in Figure 3, the implementation is structured in multiple stages:

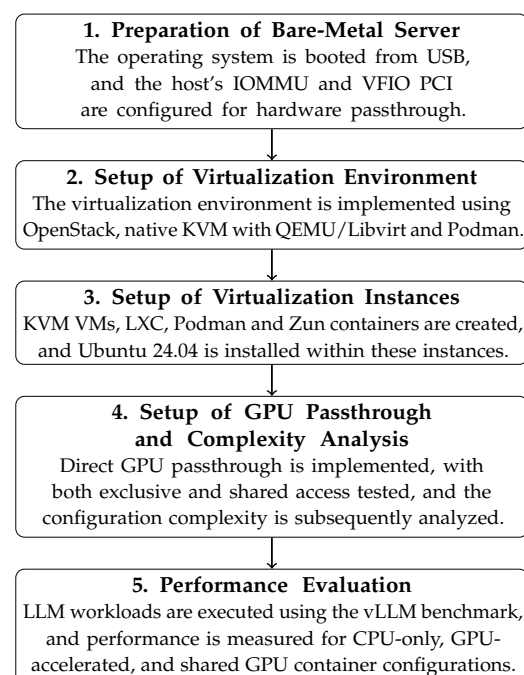


Figure 3. Five-Stage GPU passthrough setup with vLLM benchmarking.

- 1. Preparation of Bare-Metal Server:** Initially, the operating systems, Proxmox Virtual Environment (PVE) version 8.4.5 and Rocky Linux 9.6 (Kernel version 5.14.0-570.39.1.el9_6.x86_64), are booted from USB, and all necessary server setup procedures are carried out, including the configuration of IOMMU and VFIO PCI on the host (see Section 1.1).
- 2. Setup of Virtualization Environment:** For the LLM benchmark tests, multiple virtualization technologies are employed. Proxmox Virtual Environment (PVE) version 8.4.5 serves as the primary virtualization infrastructure, enabling the creation of both LXC containers and KVM-based virtual machines. Rocky Linux 9.6 is provisioned

on the server to enable deployment of the KVM hypervisor and OpenStack (version 2025.1). Beyond this, Podman (version 5.4.0) is deployed on the host as a containerization platform.

3. **Setup of Virtualization Instances:** Within the virtualization environments, individual instances, such as KVM-based VMs and containers, are created, on which the GPU is subsequently passed through and the LLM workload benchmark is executed. This encompasses Proxmox KVM VMs, Proxmox LXC containers, OpenStack KVM VMs, OpenStack Zun containers, Podman containers and native KVM VMs managed with QEMU/Libvirt. Subsequently, within the VMs and containers, Ubuntu 24.04 is used as the operating system, ensuring compatibility and usability for vLLM benchmark tests in both CPU-only and GPU-accelerated configurations.
4. **Setup of GPU Passthrough and Analysis of Configuration Complexity:** The direct GPU passthrough mechanisms for VMs and containers are implemented as described in Section 1.1. However, these instances are deployed in two configurations: one without GPU passthrough and one with exclusive access to an NVIDIA RTX 4500 Ada Generation GPU. Shared GPU usage is additionally tested in containerized environments, where two engines share one GPU at the framework level (each capped at 45% of GPU memory via vLLM's `--gpu-memory-utilization` flag, with compute time-sliced; see Section 4.2). Hardware partitioning was not used: Multi-Instance GPU (MIG) is not available on the Ada-generation RTX 4500, because MIG is offered only on selected high-end data-center GPUs (for example, the A100, A30, H100, and H200) and not on professional or workstation cards such as the RTX Ada series [13]; NVIDIA vGPU, which may be available on this card under a separate license, was not licensed or evaluated in this study [12]. Moreover, GPU access is enabled using the proprietary NVIDIA driver (version 570.172.08), obtained from the official NVIDIA source. Consequently, the configuration complexity involved establishing the virtualization platforms, particularly regarding GPU passthrough within the individual instances.
5. **Performance Evaluation:** For performance evaluation, LLM workloads are employed using vLLM benchmark tests, as LLMs have become increasingly relevant and integrated into enterprise workflows. Specifically, vLLM is a high-performance inference and serving engine for large language models, along with the Python3 benchmarking script `benchmark_serving.py`. The latter was downloaded from the official vLLM GitHub repository (version 0.10.1, <https://github.com/vllm-project/vllm/tree/releases/v0.10.1>, accessed on 25 June 2026). vLLM is optimized for efficient token generation to achieve low latency and high throughput. These characteristics make vLLM useful for both offline inference and real-time deployment in production environments [34]. Furthermore, vLLM is efficient in multi-tenant settings, where different users may simultaneously require access to distinct models [34]. For testing purposes, this process uses a relatively small, openly available model, KingNish/Qwen2.5-0.5b-Test-ft (a fine-tuned Qwen2.5-0.5B variant on the Hugging Face Hub). A small model was chosen deliberately: it keeps the workload within the single-GPU memory budget, and because every platform runs the identical model weights under a fixed seed, the platform comparison is insensitive to the specific checkpoint. The benchmarks were executed with vLLM 0.10.1 (build 0.10.1.dev497+g1bf5e1f25; CUDA 12.8, NVIDIA driver 570.172.08), using the `benchmark_serving.py` script from the vLLM repository against the OpenAI-compatible `/v1/completions` endpoint. Generation used greedy decoding (temperature 0, top-*p* 1.0, top-*k* 0, max_tokens 128) with a fixed seed of 0, confirmed in the engine configuration and the per-request sampling parameters logged at run time. The four repeated runs were therefore deterministic and produced

an identical total of 12,776 generated tokens, so the reported run-to-run variation reflects system rather than input or sampling variability.

4. Results

4.1. Analysis of Virtualization and Bare-Metal System Performance

In the evaluation, each experiment is repeated four times ($n = 4$); for every metric we report the mean and standard deviation across runs. Because every run uses a fixed seed and identical, deterministically decoded inputs (Section 3), the reported standard deviation quantifies run-to-run system variability (scheduling, context-switching, and measurement noise) and not sensitivity to input or sampling variation; input-dependent variability is out of scope for the present fixed-input design and is identified as future work in Section 5.2. Cross-platform equivalence under GPU acceleration is assessed with Two One-Sided Tests (TOST) [42,43] against an a priori $\pm 5\%$ equivalence margin. We adopt $\pm 5\%$ as an operator-defined threshold for a practically negligible difference in inference throughput, fixed before analysis rather than derived from the data; it is comfortably below both the performance gap at which an operator would reasonably change platforms and the low-single-digit-percent passthrough overheads reported for PCI passthrough in prior work [3,8], and is several times the observed run-to-run coefficient of variation ($\approx 0.9\%$). Shared-versus-exclusive throughput is compared using Welch's t -test. All per-run measurements, together with the analysis script, are provided in the openly available dataset. Notably, four inference key metrics are used for the evaluation because they provide a meaningful assessment of inference performance. As previously mentioned, the analysis highlights the metrics of latency and throughput, which include the following [34,35]:

- **Benchmark Duration (s):** Indicates the total duration of the test, providing a meaningful basis for comparing CPU and GPU performance.
- **Throughput (tok/s):** Represents the number of tokens processed per second across all active requests. Two types are distinguished: input tokens processed per second and output tokens generated per second. In this study, the output measure is employed; input (prefill) throughput, governed by the fixed 64-token prompts, is included in the released dataset.
- **Mean Time to First Token (TTFT) (ms):** Measures the time required to generate the first token after submitting a request, indicating how quickly the model can begin responding.
- **Mean Inter-Token Latency (ITL) (ms):** Captures the time interval between the generation of consecutive tokens, also referred to as Time per Output Token (TPOT). A lower ITL value implies faster token generation and, consequently, higher throughput.

The benchmark was configured with the model KingNish/Qwen2.5-0.5b-Test-ft, using 100 prompts, a maximum concurrency of 5, random input and output lengths of 64 and 128 tokens respectively, and a random dataset. We report mean TTFT and ITL in the main text; the corresponding median and p99 latencies are provided in the released dataset.

In the initial scenarios, both VMs and containers perform LLM inference using exclusive GPU access, with the entire GPU available to a single engine. These results are compared to CPU-only execution in the same instances. Performance across all virtualization instances is expected to match bare-metal systems closely because RAM and CPU have minimal impact when a GPU is used exclusively. Furthermore, GPU-accelerated execution is expected to outperform CPU-only runs substantially.

Figure 4 illustrates the results of the first scenario, comparing CPU and GPU execution time. The results are based on the mean values of four test runs.

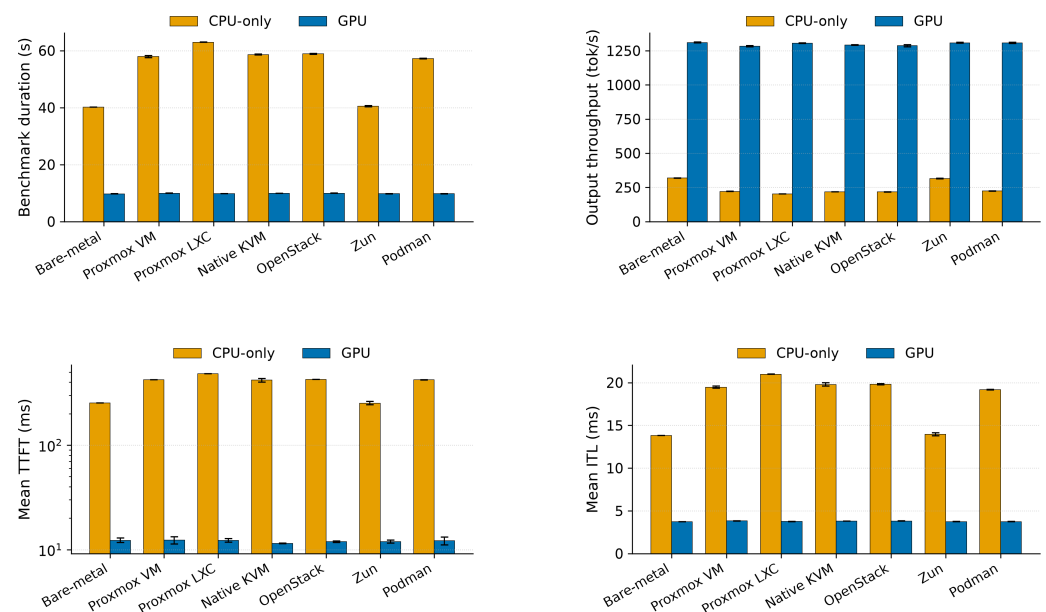


Figure 4. KPI benchmark results (GPU-accelerated, exclusive passthrough), mean $\pm$ SD over four runs: duration, throughput, mean TTFT, and mean ITL.

With an exclusive GPU, inference performance is effectively independent of the virtualization platform. Output token throughput ranges from 1283.2 ± 4.3 tok/s (Proxmox VM) to 1309.7 ± 3.6 tok/s (bare-metal server): a maximum spread of only 2.0% between the extreme system means, and a coefficient of variation of just 0.86% across the seven system means (the six virtualization platforms plus the bare-metal baseline; Table 2). The most direct, assumption-free summary is that all 28 individual runs (four runs on each of the seven systems, the bare-metal baseline included) lie within the $\pm 5\%$ band around the bare-metal mean. This is confirmed inferentially by Two One-Sided Tests (TOST) against the $\pm 5\%$ equivalence margin, using Welch's unequal-variance standard error for consistency with the shared-versus-exclusive comparison in Section 4.2: every virtualized platform is statistically equivalent to the bare-metal baseline (largest $p = 1.5 \times 10^{-4}$, for OpenStack VM; $p \leq 5.6 \times 10^{-6}$ for all others). Equivalently, each platform's 90% confidence interval for the difference from bare metal (the two one-sided 95% TOST bounds, computed with the Welch standard error) lies entirely within the ± 65.5 tok/s ($\pm 5\%$) margin, with interval half-widths of only 4.2–8.5 tok/s, so the equivalence test is highly powered despite $n = 4$ (For completeness, a one-way ANOVA across the seven systems ($df = 6.21$) does reject the null of identical means ($F = 28.0$, $p < 10^{-8}$, $\omega^2 = 0.85$). This reflects only how reproducible the per-platform timing means are at $n = 4$, not a substantive performance difference: the full spread is 2.0%, well within the $\pm 5\%$ margin, so the effect is statistically detectable yet practically negligible. Levene's median-centred test does not reject variance homogeneity ($p = 0.82$, though its power is limited at $n = 4$ per group), so the Welch standard error is a conservative choice rather than a necessity, and an equal-variance formulation yields the same conclusions. No multiple-comparison correction was applied; because the equivalence p -values reach 10^{-6} , a Holm or Bonferroni adjustment would not change any conclusion). Because every run used a fixed seed and identical greedy-decoded inputs, these four repeats quantify the systemic run-to-run noise floor under a single input realization rather than sensitivity to prompt or length variation; the equivalence claim is therefore conditional on this workload. The platforms are thus not numerically identical, but their throughput lies within a few percent of bare metal and is statistically equivalent for practical purposes.

Table 2. GPU-accelerated (exclusive passthrough) inference performance across platforms (mean $\pm$ SD over $n = 4$ runs). Cross-platform differences are within 2% of bare metal and statistically equivalent (Welch TOST, $\pm 5\%$ margin, largest $p = 1.5 \times 10^{-4}$).

Platform	Duration (s)	Output (Tok/s)	Mean TTFT (ms)	Mean ITL (ms)
Bare-metal server	9.76 $\pm$ 0.03	1309.7 $\pm$ 3.6	12.31 $\pm$ 0.61	3.74 $\pm$ 0.01
Proxmox VM	9.96 $\pm$ 0.03	1283.2 $\pm$ 4.3	12.32 $\pm$ 1.02	3.82 $\pm$ 0.01
Proxmox LXC	9.79 $\pm$ 0.02	1305.1 $\pm$ 2.4	12.29 $\pm$ 0.48	3.75 $\pm$ 0.01
Native KVM	9.88 $\pm$ 0.02	1292.6 $\pm$ 2.0	11.51 $\pm$ 0.08	3.80 $\pm$ 0.01
OpenStack VM	9.93 $\pm$ 0.05	1286.5 $\pm$ 7.3	11.95 $\pm$ 0.20	3.81 $\pm$ 0.02
Zun container	9.77 $\pm$ 0.02	1308.1 $\pm$ 2.7	11.96 $\pm$ 0.39	3.74 $\pm$ 0.01
Podman container	9.78 $\pm$ 0.03	1306.7 $\pm$ 4.6	12.15 $\pm$ 1.07	3.75 $\pm$ 0.01

Table 3 makes this pairwise structure explicit, giving each platform's throughput difference from bare metal, its 90% confidence interval (the two one-sided 95% TOST bounds, computed with the Welch standard error), and the equivalence p -value. Every interval lies inside the ± 65.5 tok/s ($\pm 5\%$) margin, yet the point differences are not zero: Proxmox VM sits 2.0% below bare metal with a confidence interval that excludes zero, whereas Zun and Podman differ by less than 0.3% and their intervals include zero. What the equivalence test certifies is therefore that each platform lies within the $\pm 5\%$ margin, not that the platforms are numerically identical; the residual differences are real, and the one-way ANOVA above resolves them, but they are practically negligible. This holds only for the operating mode tested. As Section 5 develops, it should not be expected to survive once the GPU is saturated, the request rate rises substantially, or multi-GPU topologies are introduced, where per-platform scheduling and interconnect behaviour can differ.

Table 3. Per-platform output-throughput difference from bare metal under exclusive GPU passthrough (tok/s, $n = 4$). The 90% confidence interval is formed from the two one-sided 95% TOST bounds using the Welch standard error; the $\pm 5\%$ equivalence margin is ± 65.5 tok/s. Every interval lies entirely within the margin, so each platform is statistically equivalent to bare metal, while the non-zero point differences confirm the platforms are not numerically identical.

Platform	Diff. from Bare Metal (Tok/s)	Diff. (%)	90% CI (Tok/s)	TOST p
Proxmox VM	−26.5	−2.02	[−32.0, −21.0]	5.6×10^{-6}
Proxmox LXC	−4.6	−0.35	[−8.9, −0.3]	3.4×10^{-7}
Native KVM	−17.1	−1.30	[−21.3, −12.8]	2.2×10^{-6}
OpenStack VM	−23.2	−1.77	[−31.6, −14.7]	1.5×10^{-4}
Zun	−1.6	−0.12	[−6.0, +2.9]	1.5×10^{-7}
Podman	−3.0	−0.23	[−8.7, +2.7]	5.9×10^{-7}

The remaining metrics follow the same pattern. GPU-accelerated execution is roughly five- to six-fold faster than CPU-only execution for the 24-core-matched instances, and about four-fold for the bare-metal and Zun baselines, whose CPU-only runs were not restricted to 24 cores; benchmark duration under GPU acceleration is near-identical across platforms (9.76–9.96 s; Table 2). For mean TTFT, native KVM (11.51 $\pm$ 0.08 ms) and OpenStack (11.95 $\pm$ 0.20 ms) are nominally lower than the bare-metal server (12.31 $\pm$ 0.61 ms); these sub-millisecond gaps fall well within the run-to-run standard deviation and should be read as measurement noise rather than a genuine advantage over bare metal. Mean ITL and TPOT are likewise stable across platforms under GPU acceleration. Extending the same equivalence test to the other metrics, benchmark duration and mean ITL are statistically equivalent to bare metal for every platform (90% confidence intervals within the $\pm 5\%$ margin); mean TTFT is the exception, because its $\pm 5\%$ margin (± 0.62 ms) is smaller than the run-to-run variation at $n = 4$, so TTFT equivalence cannot be formally certified even though no platform departs from bare metal beyond noise.

The Zun container is treated as a special case for the CPU-only comparison. Although nominally limited to 24 cores, Zun's "shared" CPU policy grants it access to all host cores

(Even when 24 CPU cores are explicitly assigned, Zun still grants access to all 96 logical host threads by default (the 48-core EPYC 9454 exposes 96 threads via SMT). The CPU allocation policy could not be changed from “shared” to “dedicated” within the OpenStack/Zun configuration, as the relevant setting is undocumented), so its CPU-only results, which approach the bare-metal baseline, are not comparable to the 24-core instances. They are therefore excluded from the CPU-only cross-platform comparison and annotated as an outlier. Under GPU acceleration this confound is immaterial, because CPU and RAM have negligible influence when the GPU is used exclusively.

In summary, when GPU acceleration is applied, LLM inference performance is statistically equivalent across all evaluated virtualization technologies and closely matches bare metal; the only large effect is the roughly fourfold gain of GPU over CPU execution. Within the tested regime, a single passthrough GPU, a small 0.5B-parameter model, and a maximum concurrency of five, the choice of virtualization platform does not meaningfully affect GPU-accelerated inference performance. The scope and limits of this finding are examined in Section 5.

4.2. Shared GPU Workload Execution in Containerized Environments

In this section, two container instances execute LLM inference concurrently on the same physical GPU. Sharing was implemented at the framework level: each vLLM engine was capped at 45% of GPU memory via the `--gpu-memory-utilization 0.45` flag; because vLLM statically pre-allocates a KV-cache block pool sized by this fraction at start-up, the cap is what lets two engines coexist within the 24 GiB of VRAM. Each engine loaded its own independent copy of the model weights (there is no weight sharing across the two CUDA contexts), so the 45% cap bounds each engine’s combined weight and KV-cache footprint. The GPU’s compute resources were not partitioned; both engines shared the streaming multiprocessors through the default time-slicing of CUDA contexts, without Multi-Instance GPU (MIG) or the Multi-Process Service (MPS) [29]. The 45% cap therefore bounds only each engine’s memory footprint so that both fit in 24 GiB of VRAM; it provides no compute isolation or quality-of-service guarantee, because the streaming multiprocessors are time-sliced between the two CUDA contexts. This memory-fraction sharing is a deliberate software configuration and is therefore independent of vGPU support; it was applied to containers, where running two concurrent engines is straightforward. Only GPU execution is considered, compared against exclusive GPU usage for the same container technology.

The initial hypothesis was that the aggregate performance of two concurrent engines would simply equal that of a single engine with exclusive GPU access, redistributed across the two containers, that is, roughly half the throughput each, with no net gain.

The two concurrent engines achieved nearly identical performance to each other (Figure 5), but the “no net gain” hypothesis was not confirmed. Instead, aggregate throughput under sharing exceeded exclusive single-engine throughput by a consistent margin: +14.4% for Proxmox LXC ($1305.1 \rightarrow 1493.3 \pm 31.8$ tok/s), +12.3% for Zun ($1308.1 \rightarrow 1469.5 \pm 9.0$ tok/s), and +12.3% for Podman ($1306.7 \rightarrow 1467.5 \pm 5.9$ tok/s). All three gains are statistically significant (Welch’s *t*-test, $p \leq 1.2 \times 10^{-3}$; Table 4), though with $n = 4$ per condition the Welch degrees of freedom are small (≈ 3 – 6) and the *p*-values should be read as indicative. Per-engine throughput roughly halved (to ≈ 733 – 749 tok/s) while per-token latency roughly doubled (mean ITL $3.7 \rightarrow 6.6$ – 6.7 ms), consistent with two engines time-sharing the same streaming multiprocessors.

Three explanations are consistent with this gain, and the aggregate-throughput data alone cannot distinguish them; we list them from most to least likely given the setup. First, because the two concurrent engines are driven by two independent benchmark clients (each at a maximum concurrency of five), the aggregate offered load is roughly double

that of the single-engine baseline, so part of the gain may reflect higher offered load rather than any property of GPU sharing. Second, host-side latency hiding: kernel-launch and request-scheduling gaps that leave the device idle between kernels for a single engine are overlapped when a second engine runs concurrently. Third, a single 0.5B-parameter engine at a concurrency of five may leave streaming-multiprocessor capacity idle; under the default time-slicing scheduler, however, the two CUDA contexts are predominantly interleaved rather than co-executed spatially, so genuine spatial recovery of idle SMs would require the Multi-Process Service (MPS) and is unlikely to be the dominant factor here. Because offered load was not held constant, the gain is consistent with (but does not by itself establish) GPU under-utilization; disentangling these mechanisms would require a load-matched single-engine control (one engine at `--max-concurrency 10`) together with direct GPU-utilization telemetry (SM occupancy, memory bandwidth, and power draw), both left to future work (Section 5.2). We therefore report the gain descriptively and leave its mechanism open. The small TTFT variations between paired containers, most visible for Zun and Podman, are within run-to-run noise and are attributable to scheduling and context-switching effects under concurrent execution.

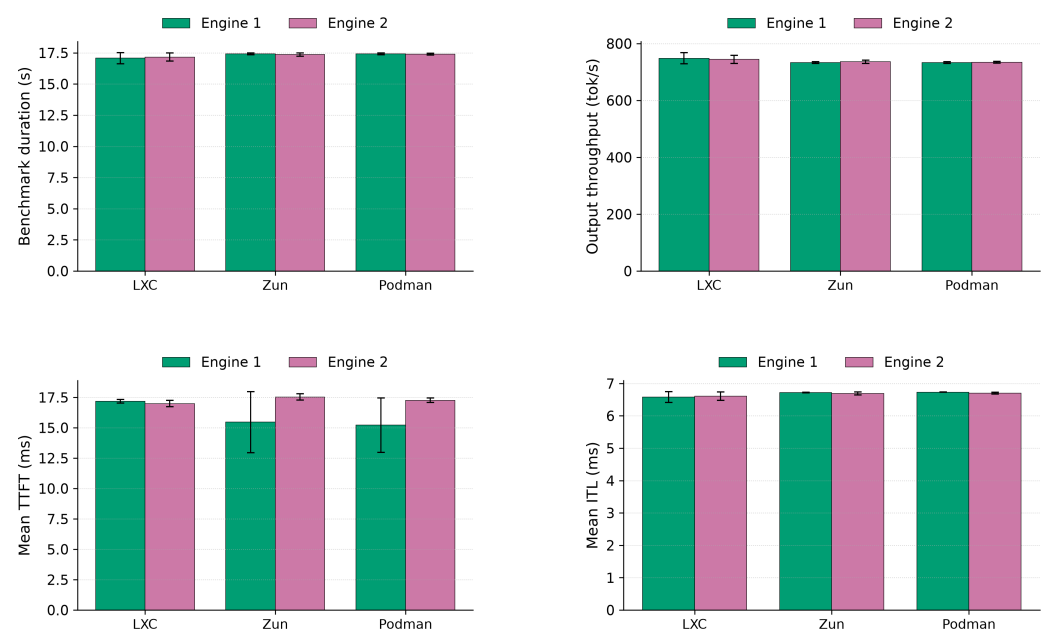


Figure 5. Parallel (shared) GPU execution, mean $\pm$ SD over four runs: duration, throughput, TTFT, and ITL.

Table 4. Shared GPU execution (two concurrent vLLM engines, each capped at 45% of GPU memory) versus exclusive single-engine execution. Aggregate output throughput is the sum of both engines, mean $\pm$ SD over $n = 4$ runs. The shared condition also doubles the offered load (two independent clients), so the gain is reported descriptively and does not by itself isolate a GPU-sharing benefit (Section 4.2); with $n = 4$ the Welch degrees of freedom are small (≈ 3 – 6) and the p -values are indicative only.

Technology	Exclusive (Tok/s)	Shared Aggregate (Tok/s)	Gain	Welch t (df), p
Proxmox LXC	1305.1 ± 2.4	1493.3 ± 31.8	+14.4%	11.8 (3.0), 1.2×10^{-3}
Zun	1308.1 ± 2.7	1469.5 ± 9.0	+12.3%	34.3 (3.5), 1.4×10^{-5}
Podman	1306.7 ± 4.6	1467.5 ± 5.9	+12.3%	42.9 (5.6), 2.5×10^{-8}

4.3. Overview of Implementation Complexity with Emphasis on GPU Passthrough

Overall, GPU passthrough was successfully implemented across the selected virtualization technologies and, more specifically, on the individual instances. However, the

mechanisms involved and the overall setup process differ between technologies, producing varying levels of configuration complexity. More precisely, whereas setting up Proxmox VMs, LXC containers and the native KVM VM on bare-metal Linux using QEMU/Libvirt with GPU passthrough was relatively straightforward, configuring the OpenStack VM, particularly the Zun container service with GPU passthrough, proved to be significantly more complex than anticipated. Specifically, Podman requires the creation of a CDI specification file, which also differs from the procedures used for the other technologies.

To make the comparison systematic rather than purely narrative, Table 5 rates each platform on four qualitative criteria derived from the deployment experience: host preparation, the number of passthrough configuration steps, the number of distinct services or components that must be coordinated, and the gaps in usable documentation (which drive trial-and-error effort). Ratings are ordinal (Low, Moderate, High, Very High), reflect relative effort, and were assigned by the single team that performed all deployments; absolute setup times, step counts, and the number of edited configuration files were not recorded systematically and are noted as a limitation (Section 5.2).

These ratings capture the total first-deployment effort, which for the multi-service platforms bundles two separable costs. The first is the baseline effort of standing up the platform itself, independent of any GPU: for OpenStack this is the deployment of the IaaS control plane via Kolla-Ansible, which dominates its total rating. The second is the incremental effort of enabling GPU passthrough on top of an already-running platform: for OpenStack and Zun this is the Nova PCI alias, whitelist, filter, and GPU-flavor configuration (plus, for Zun, privileged mode and device injection). Judged on the incremental passthrough step alone, OpenStack VM is closer to Moderate; its High/Very High total rating reflects control-plane deployment and its undocumented configuration surface more than the passthrough mechanism itself. Native KVM is the mirror image: as detailed in the KVM discussion below, once the GPU is bound to VFIO on the host, its passthrough step is comparable in effort to Podman's, and its elevated rating reflects VM-provisioning ergonomics rather than passthrough. The single-node platforms (Proxmox, native KVM, Podman) have a negligible baseline beyond the host operating system, so for them total and incremental complexity nearly coincide. Table 5 should therefore be read as total first-deployment effort; the passthrough increment alone is more uniform across platforms than the totals suggest, with Zun the clearest exception because its privileged-mode requirement and undocumented device-injection path are intrinsic to the passthrough step rather than to the IaaS baseline.

Table 5. Configuration-complexity rubric for enabling GPU passthrough on each platform. Ordinal ratings assigned by the single team that performed all deployments (a structured first-deployment experience report rather than a multi-rater measurement); higher indicates greater effort. The rank order is the reliable signal; absolute levels are indicative.

Platform	Host Prep.	Config. Steps	Services to Coordinate	Doc. Gaps	Overall
Proxmox LXC	Moderate	Low	Low	Low	Low
Proxmox VM	Moderate	Low	Low	Low	Low
Podman	Low	Moderate	Low	Low	Low–Moderate
Native KVM	Moderate	High	Moderate	Moderate	Moderate
OpenStack VM	Moderate	High	Very High	Moderate	High
Zun (OpenStack)	Moderate	Very High	Very High	Very High	Very High

To make these ordinal ratings traceable and to reduce reliance on subjective judgement, Table 6 reports the countable artifacts underlying each level, reconstructed from the deployment records. We distinguish two kinds of anchor. The structural counts (platform-specific configuration files edited, distinct services that must be coordinated, and host-preparation steps) are objective and largely familiarity-invariant: they follow from each platform's

architecture and would look similar for any competent operator, so they form the primary, reproducible evidence for the rank order. The experience-sensitive figures (wall-clock setup time and trial-and-error iterations) depend on the operators' prior familiarity and are reported only as order-of-magnitude, first-deployment estimates. Because all deployments were carried out by a single team, this analysis is presented as a structured first-deployment experience report rather than a controlled measurement; a multi-operator study, with at least three independent operators and prospectively logged setup time, step counts, and manual commands, would yield inter-rater variability and is the natural next step (Section 5.2).

Table 6. Indicative anchors for the configuration-complexity ratings in Table 5, reconstructed retrospectively from a single team's first-deployment records. The platform-specific config files column lists files edited beyond the host preparation common to the VFIO-based platforms (GRUB/IOMMU and modprobe edits), which is captured separately in the host-prep steps column; together with services coordinated, these reflect structural requirements that are largely familiarity-invariant (they would look similar for any competent team). The setup time (given as order-of-magnitude estimates, marked ~) and trial-and-error iterations columns are experience-sensitive: a team already familiar with a given platform would see lower values, so treat these as illustrative first-deployment estimates, not controlled measurements.

Platform	Platform-Specific Config Files	Services Coordinated	Host-Prep Steps	Setup Time (h)	Trial-and-Error Iters.
Proxmox LXC	<code>/etc/pve/lxc/*.conf</code>	PVE System, LXC Service, <code>nvidia.ko</code> module	Unbind VFIO, install host NVIDIA drivers, blacklist open-source drivers, and configure device node forwarding via <code>cgroup</code> allow rules and bind mounts for <code>/dev/nvidia*</code>	~2	1
Proxmox VM	None (GUI)	PVE System, <code>qm</code> QEMU/KVM Virtual Machine Manager, QEMU/KVM Process Layer, VFIO Framework	Enable BIOS/GRUB IOMMU, blacklist open-source host drivers, configure VFIO modules, and attach the physical GPU (IOMMU group) directly via the PVE WebUI	~2	1
Podman	<code>cdi/nvidia.yaml</code>	Podman Engine, NVIDIA Container Toolkit, CDI	Unbind VFIO, install host NVIDIA drivers and Toolkit, generate CDI specification, and pass GPU via <code>-device nvidia.com/gpu=all</code> (CDI)	~2	1
Native KVM	None (CLI)	Libvirt Daemon, QEMU Emulator, VFIO Framework	Enable BIOS/GRUB IOMMU, blacklist open-source drivers, configure VFIO modules, and deploy VM via <code>virt-install</code> with <code>-host-device</code> flags	~3	3
OpenStack VM	<code>globals.yaml</code> , <code>/etc/kolla/config/nova/*</code>	Nova (API, Compute, Scheduler with <code>PciPassthroughFilter</code>), Placement, Glance, Neutron, Keystone, Cinder, RabbitMQ, MariaDB, QEMU/KVM, VFIO Framework	Enable BIOS/GRUB IOMMU, blacklist open-source drivers, configure VFIO, deploy OpenStack via Kolla-Ansible, and configure Nova aliases, whitelists, filters, and GPU flavors	~8	5
Zun (OpenStack)	<code>/etc/kolla/config/zun/*</code>	Zun, plus the shared OpenStack control plane it depends on (Nova/Placement, Keystone, Glance, Neutron, RabbitMQ, MariaDB) and Kuryr/CNI	Unbind VFIO, install host NVIDIA drivers, add Zun overrides (aliases, whitelists, filters) via Kolla, and execute service redeployment	~12	8

Proxmox VM and LXC

The setup of the Proxmox virtualization environment is relatively straightforward because it is installed directly as the operating system on the bare-metal server. In this vein, VMs and LXC containers can be configured intuitively via the graphical web interface. Additionally, GPU passthrough requires certain preparations on the host system, particularly enabling IOMMU and VFIO-PCI as described in Section 1.1. For LXC containers, the proprietary NVIDIA driver must be installed on the host. In VMs, passthrough is configured via the hardware section by selecting the PCI device ID, enabling all associated functions, and installing the proprietary NVIDIA driver directly within the VM. In contrast, LXC

containers do not require device assignment through the interface; they can be created in unprivileged mode with only NVIDIA user-space components installed.

OpenStack

In contrast to Proxmox and native KVM, GPU passthrough in OpenStack requires a significantly more complex configuration. That is, this entity must be integrated across multiple Nova services, including Nova-API, Nova-Compute and Nova-Scheduler. This process involves defining a GPU alias, creating a whitelist and configuring PCI filters to enable passthrough. Moreover, the alias must be assigned using a flavor, which is subsequently used by the instance to gain access to the GPU. Furthermore, the Nova services must be redeployed because the default configuration is modified by Kolla overrides. It is important to allocate GPUs in OpenStack via flavors, implying that the corresponding resources are reserved within the project. Consequently, once a GPU is assigned to a VM, it cannot be reallocated to another instance without first deleting or reconfiguring the original VM, a significant limitation in resource flexibility. Frequently, successful deployment necessitates troubleshooting because issues may arise due to the complex interdependencies of the underlying services, and the causes of these problems are not immediately apparent. Overall, the general setup of OpenStack is the most demanding because it involves numerous preparatory steps and a time-consuming deployment process using Kolla-Ansible. Nevertheless, extensive documentation and support resources are available, which can be leveraged to ensure proper GPU passthrough.

KVM

After establishing Proxmox, the server is provisioned with Rocky Linux 9.6 to install the KVM (kernel-based) hypervisor. Because KVM is already integrated as a kernel module, only the QEMU and Libvirt packages must be installed, with VM creation performed via the command-line tool `virt-install`, preferred in this study. For proper installation, specifying the kernel parameters `kernel=casper/vmlinuz` and `initrd=casper/initrd` is recommended to avoid stalling in a serial, text-based console. For comparability in CPU-only benchmarking explicit CPU settings `-vcpus=24 sockets=1,cores=24,threads=1` must be applied to prevent impractical default allocations. As with Proxmox, GPU passthrough requires preparing the host by enabling IOMMU and configuring VFIO-PCI described in Section 1.1. In this setup the GPU is already bound through VFIO on the host. Thus, identifying the GPU and its associated audio device with `virsh nodedev-list` and passing them via `-host-device` ensures proper passthrough. Using the q35 machine type and installing the appropriate driver in the VM enables immediate GPU access. In this procedure, VM setup requires a thorough procedure due to potential pitfalls and limited guidance, such as misconfigured vCPUs or kernel parameters. Much of this friction, however, stems from general unattended-install and `virt-install` ergonomics rather than from GPU passthrough itself: once the GPU is bound to VFIO on the host, the passthrough step is comparable in effort to Podman's, so native KVM's higher complexity rating reflects VM provisioning as much as passthrough. However, GPU passthrough is readily configured through the use of the `-host-device` parameters, allowing direct assignment to the VM.

Zun Container Service

Zun, the OpenStack container service, is particularly complex with no established approach for the proper deployment of containers within OpenStack, especially regarding GPU passthrough. Whereas Podman and LXC permit unprivileged containers to access the GPU, extensive research and practical testing demonstrated that Zun containers require privileged mode to access the `/dev/nvidia*` directory for GPU passthrough. We were unable to enable the targeted device-injection path used for LXC and Podman (explicitly passing `/dev/nvidia0`, `/dev/nvidiactl`, and `/dev/nvidia-vm` with `cgroup allow` rules) through Zun's documented configuration, so privileged mode was the only route we

found; whether the unprivileged targeted path is achievable in Zun remains open and may reflect the same undocumented configuration surface noted below. Thereafter, deploying only the userspace components of the proprietary NVIDIA driver within the container is sufficient to enable GPU access. As noted in the exclusive GPU scenario, resource usage in Zun is skewed, which compromises the comparability of pure CPU benchmark results. This issue arises from resource policies, as highlighted in the preceding section, making it extremely difficult to change the CPU allocation policy from “shared” to “dedicated” (see Section 4.1). This behavior contrasts with Podman and LXC, where resource limits are strictly enforced as configured. Not only the CPU policy, but each Zun container is also constrained by a fixed hard limit, set below the required resources of 24 cores and 32 GiB RAM. Raising this hard limit is cumbersome because it is not immediately apparent in which schema or configuration file the adjustment should be made. In some cases, changes could be applied only through Kolla overrides. Overall, the setup required significantly more effort, extensive research, and practical testing due to insufficient documentation. Similarly, maintenance is also relatively arduous because containers automatically stop after a period of inactivity.

Podman

Finally, Podman offers a straightforward setup via the package manager and is immediately ready for use. Next, GPU passthrough, while differing from hypervisor-based approaches, is not overly complex. As mentioned in 1.1, the NVIDIA user-space components within the container for the benchmark scenarios are sufficient. The approach applied for LXC and Zun also functions for Podman. However, Container Device Interface (CDI) integration and the container toolkit are generally recommended for GPU passthrough in Podman. Therefore, the CDI specification on the host (`/etc/cdi` or `/var/run/cdi`) is configured, along with the installation of the NVIDIA Container Toolkit. This procedure allows Podman to identify and expose the GPU to containers, with passthrough achievable via `-device nvidia.com/gpu=all`. This study used the specialized `nvidia/cuda:12.6.1-base-ubuntu24.04` image, following best practices. Podman’s host preparation is comparatively light because, unlike the VFIO-based platforms, it requires no IOMMU/GRUB reconfiguration or rebinding of the GPU to `vfio-pci`: the GPU remains on the NVIDIA host driver, and only the driver and Container Toolkit are installed. The container’s CUDA 12.6 user-space is forward-compatible with the CUDA 12.8-capable host driver (version 570.172.08).

Although GPU passthrough in Proxmox LXC and KVM VMs is relatively straightforward, supported by extensive documentation and an intuitive web interface, the process is considerably more complex in the Zun Container Service. Due to limited documentation and the reliance on trial-and-error methods, GPU passthrough in Zun is the most demanding among the evaluated technologies. There is little guidance on how to pass a GPU to a container, but extensive research and practical testing finally enabled a successful implementation.

5. Discussion

These experiments demonstrated that the performance with exclusive GPU utilization in the virtualized environments tested is nearly identical to that of a bare-metal server. Thus, these results indicate that modern virtualization technologies can efficiently leverage GPUs when direct passthrough is employed. Consequently, this observation aligns with theoretical expectations, which assert that direct GPU passthrough offers the principal advantage of near-native performance and low-latency, transparent access to GPU resources. Conversely, the limitation imposed by exclusive resource allocation is corroborated by the literature because a GPU assigned to a single VM cannot be concurrently allocated

to another VM without first terminating and reconfiguring the original instance. This exclusivity is a property of PCI passthrough itself; the framework-level memory-fraction sharing examined in Section 4.2 is a distinct mechanism that does not rely on passthrough and is therefore not subject to the same constraint. Notably, this restriction is pronounced in OpenStack, where releasing a GPU for allocation to another VM necessitates deleting the original instance because underlying flavors and project resources are pre-allocated and reserved, potentially constraining flexible resource management.

Across all benchmarks, a consistent trend was observed. For instance, GPU-accelerated performance outperformed CPU-only execution by roughly four- to six-fold, demonstrating the expected computational advantages of GPU acceleration. Equally important, the results indicate that performance differences among the virtualization technologies evaluated (Proxmox VM and LXC, native KVM, OpenStack VM, Zun, and Podman) were minimal when the GPU was exclusively assigned. Thus, under this GPU-accelerated workload, inference performance is largely independent of the underlying hypervisor or container runtime environment. Additionally, these findings are consistent with previous studies reporting near-native GPU performance under PCI passthrough configurations [3,8,28], which documented passthrough overheads in the low single-digit percent; the $\leq 2\%$ cross-platform spread observed here sits within that envelope. This outcome is expected rather than surprising: all six platforms route GPU access through the same low-level passthrough path: VFIO/KVM for the VM-based platforms (Proxmox VM, native KVM, OpenStack VM) and host-kernel cgroup device rules (optionally declared through the Container Device Interface specification) for the container-based platforms (Proxmox LXC, Zun, Podman); so under a non-saturating workload this equivalence is best read as a confirmatory baseline. Whether it persists once the GPU is saturated, or under multi-GPU topologies where interconnect and scheduling behaviour differ across platforms, remains open, and the result may not merely weaken but invert. The overheads most likely to surface at scale are IOMMU and interrupt-remapping pressure and vCPU-scheduling jitter under high request rates (which affect tail latency), and, most sharply, multi-GPU topologies, where NVLink/PCIe peer-to-peer exposure and NUMA affinity are handled differently across Proxmox, native KVM, and OpenStack.

Notably, the Zun container service exhibited outlying behaviour in CPU-only tests because of its shared CPU allocation policy: although nominally limited to 24 cores, it accessed all logical host threads, yielding CPU-only performance close to the bare-metal baseline (the mechanism is detailed in Section 4.1). Under GPU-accelerated execution this confound is negligible, because CPU and RAM exert minimal influence when the GPU is used exclusively.

The second set of experiments focused on shared GPU utilization in containerized environments. Two containers executed LLM inference in parallel, with each limited to approximately 45% of GPU memory (compute time-sliced). In contrast to initial expectations, the aggregate performance exceeded that observed with exclusive GPU allocation by 12–14%; because the shared condition also doubled the offered load (two concurrent clients), this is consistent with, but not proof of, the single-container workload leaving GPU capacity idle. This aggregate gain, however, came with a roughly twofold increase in per-token latency (mean ITL $3.7 \rightarrow 6.6\text{--}6.7$ ms), so framework-level time-sliced sharing [44] is advisable only for throughput-oriented or batch serving within a single trust domain, and is unattractive for latency-sensitive or multi-tenant workloads bound by a service-level objective, which would instead require hardware partitioning (MIG or vGPU) or the Multi-Process Service; energy- and cost-efficiency metrics (for example, tokens per watt or per GPU-hour) that ultimately govern such decisions were not measured here.

These results highlight the need for further investigation into fine-grained GPU scheduling strategies [31,33] to optimize GPU utilization in virtualization environments.

However, the virtualization technology adopted (hypervisor- or container-based) did not meaningfully affect performance. Even so, the complexity of implementing GPU passthrough varied considerably across the evaluated platforms, potentially influencing the choice of a virtualization environment. Proxmox and native KVM allowed relatively straightforward configuration through well-documented GPU passthrough settings. On the other hand, OpenStack, particularly its Zun container service, requires extensive manual adjustments and Kolla overrides. This complexity arises from the need to coordinate multiple Nova services and resource allocation policies. In contrast, Podman offers simpler, more direct integration with the NVIDIA Container Toolkit and CDI specifications, enabling efficient GPU passthrough for containerized workloads. The privileged-mode requirement for Zun GPU access additionally carries a security cost: a privileged container receives broad host device access and weakened namespace isolation, so the reduced-isolation rating in Table 7 reflects a concrete increase in attack surface rather than a purely architectural judgement; whether this requirement is intrinsic to Zun or an artefact of its undocumented configuration path was not established and warrants further investigation.

When platform choice would matter: The equivalence reported here is bounded to the tested regime. Several scenarios in which the platforms would be expected to diverge lie outside what the present design can observe. (i) GPU saturation: once a larger model or a higher request rate keeps the device continuously busy, differences in vCPU scheduling, interrupt remapping, and IOMMU pressure across hypervisors and container runtimes can surface as tail-latency and throughput gaps that are invisible under the non-saturating load tested here. (ii) Large models (7B–70B): greater memory pressure, paging, and longer kernels change the ratio of compute to overhead and may expose passthrough or scheduling costs that are negligible for a 0.5B-parameter model. (iii) Multi-GPU topologies: NVLink and PCIe peer-to-peer exposure and NUMA affinity are handled differently across Proxmox, native KVM, and OpenStack, so multi-GPU collective performance is where platform choice is most likely to matter, and where equivalence could not merely weaken but invert. (iv) Latency-sensitive, SLO-bound, or multi-tenant serving: under contention, the isolation and quality-of-service properties that differ across platforms, and that the single-tenant workload here never exercised, become first-order. (v) Steady-state operation: recurring burdens such as Zun containers stopping after inactivity or OpenStack services requiring redeployment on configuration change (Section 5.1) accumulate over a deployment's lifetime and are not captured by one-time setup effort. These scenarios are the natural targets for the future work outlined in Section 5.2.

5.1. Platform Selection Guidance

Because GPU-accelerated performance is statistically equivalent across the evaluated platforms (Section 4.1), platform selection should be driven by operational factors rather than throughput. Table 7 synthesizes the trade-offs. Two dimensions are grounded in this study, namely GPU performance and configuration complexity (Table 5); isolation, multi-tenancy, and maintenance are qualitative assessments derived from each platform's architecture and the deployment experience, were not measured, and should be read as guidance rather than results. A central tension is that lower configuration complexity does not imply a better choice: OpenStack's complexity is largely the cost of capabilities that single-node platforms lack, namely multi-tenancy, resource quotas, and API-driven automation. For a single host serving a few models, Proxmox or Podman minimize effort; for a shared, multi-team private cloud, OpenStack's overhead may be justified despite the demanding setup. All evaluated platforms are open-source; OpenStack additionally

carries higher operational cost, and NVIDIA vGPU licensing would apply only if hardware GPU partitioning were adopted (Section 5.2). Finally, because the multi-tenant capabilities that justify OpenStack’s overhead (quotas, isolation, and API-driven scheduling under contention) were not exercised by the single-tenant workload evaluated here, the complexity findings are most directly actionable for single-host or small-team operators, whereas the guidance for large multi-tenant clouds remains comparatively provisional. The complexity ratings also capture one-time setup effort; recurring steady-state burden (such as Zun containers stopping after inactivity or OpenStack services requiring redeployment on configuration change) is a separate operational cost that this study did not quantify and that may dominate over a deployment’s lifetime.

Table 7. Platform-selection heuristics. Only the GPU-performance and setup-complexity columns are grounded in this study’s measurements and deployment; the isolation, tenancy/API, and maintenance columns are architectural judgements, not exercised under contention here, and should be read as expert opinion rather than results.

Platform	GPU Perf. ^c	Setup	Isolation ^b	Tenancy/API ^b	Maint. ^b	Typical Fit
Proxmox VM	Equivalent	Low	High	Limited	Low	Small/mid on-prem; strong VM isolation
Proxmox LXC	Equivalent	Low	Moderate	Limited	Low	Lightweight on-prem container hosting
Native KVM	Equivalent	Moderate	High	Manual	Moderate	Scripted single-host, full control
OpenStack VM	Equivalent	High	High	Strong	High	Large multi-tenant private cloud
Zun	Equivalent	Very High	Reduced ^a	Strong	High	Container-native OpenStack (with caveats)
Podman	Equivalent	Low-Mod.	Moderate	Limited	Low	Simple single-host container GPU serving

^a Zun required privileged-mode containers for GPU access, weakening isolation relative to unprivileged Podman/LXC; this rating is contingent on the configuration path we found and may not be intrinsic to Zun (Section 4.3).

^b Architectural judgement, not exercised under contention in this study, and not a measured result; the maintenance rating in particular reflects architectural component count rather than measured operational effort.

^c Equivalence was established only in the non-saturating, small-model, low-concurrency regime tested (Section 4.1); it is not claimed for saturating or hardware-partitioned workloads.

5.2. Limitations

Several boundaries of this study should be made explicit. First, the performance evaluation uses a single small language model (Qwen2.5-0.5B) at a maximum concurrency of five on a single passthrough GPU; this workload does not saturate the GPU, as the shared-execution results demonstrate. The finding that platform choice does not affect GPU-accelerated performance therefore holds for small-model, low-concurrency inference and should not be extrapolated to larger models (for example, 7B–70B parameters), higher concurrency, longer sequences, or multi-GPU configurations, where passthrough and scheduling overheads may become visible. Second, the observation that shared execution exceeds exclusive execution is explained in terms of GPU under-utilization inferred from aggregate throughput; direct utilization telemetry (streaming-multiprocessor occupancy, memory bandwidth, and power draw) was not captured and is required to confirm the mechanism. Third, GPU sharing was realized through framework-level memory capping with CUDA time-slicing rather than hardware partitioning (MIG or vGPU), so isolation and quality-of-service guarantees under contention were not evaluated; nor was the Multi-Process Service (MPS), which enables spatial rather than time-sliced co-execution on the same GPU and would be the natural baseline for separating recovered-capacity from offered-load effects in the shared experiment. Fourth, the configuration-complexity assessment draws on the authors’ hands-on deployment experience: the ordinal ratings were assigned by the single team that performed all deployments, the quantitative anchors (Table 6) were reconstructed from deployment records rather than logged prospectively, and the setup-effort figures reflect first-time deployment and therefore conflate intrinsic platform complexity with the team’s prior familiarity. Although systematized in Section 4.3, it is best read as a structured experience report rather than a controlled measurement; a

multi-rater study with prospectively logged effort is left to future work. Fifth, the steady-state control-plane resource footprint of each platform was not measured. This footprint differs substantially across platforms. A bare Podman daemon or libvirtd process consumes well under 100 MB of RAM, whereas a full OpenStack deployment (Nova, Neutron, Glance, Keystone, Cinder, and their databases) occupies several gigabytes. On memory-constrained hosts this overhead may dominate the platform-selection decision. Quantifying it is left to future work. Sixth, Podman was evaluated in its default rootful configuration; rootless Podman, which alters the device-injection and CDI path, was not assessed. Seventh, the statistical comparisons rest on four repeated runs per condition, measured on a single GPU model (the RTX 4500 Ada Generation) in one server. Although this sample size already yields high equivalence power at the observed low run-to-run variance (coefficient of variation $\approx 0.9\%$), a larger number of repetitions and, more importantly, replication on other GPU architectures (for example, data-center A100/H100 or AMD accelerators) and on multi-GPU hosts would establish how far the platform-equivalence and GPU-sharing results generalise beyond the single Ada-generation card evaluated here. These limitations define concrete directions for future work.

6. Conclusions

This study evaluated GPU passthrough across six widely used virtualization platforms under an LLM inference workload, pairing performance measurements with a configuration-complexity analysis. Its primary contribution is the structured, paired comparison of configuration complexity alongside performance (RQ3), a pairing absent from prior passthrough benchmarks. This comparison is offered as a single-team, first-deployment experience report rather than a multi-rater measurement. The pairing nonetheless yields an actionable insight that neither axis provides alone: because performance is equivalent in this regime, platform choice can be driven almost entirely by configuration and operational complexity rather than throughput, which inverts the usual performance-first selection heuristic for single-GPU, small-model serving. Among the evaluated platforms, the LXC and native-KVM arms revisit well-studied baselines under a current software stack [8]; the novel coverage lies in OpenStack Zun and Podman/CDI, two under-documented paths. Across all platforms, Proxmox (VM and LXC) and Podman proved straightforward, native KVM is moderately involved, and OpenStack, especially its Zun container service, is the most demanding. On the performance side, the results serve as a confirmatory baseline. For exclusive passthrough with a small model under light load, inference performance is statistically equivalent across all platforms and close to bare metal (RQ1), consistent with the near-zero overhead of PCI passthrough reported in prior work. Because this workload does not saturate the GPU, the equivalence is regime-bounded and should be read as a baseline rather than a claim that platform choice is irrelevant in general; it may not hold once the GPU is saturated or hardware-partitioned. Running two containers concurrently on one GPU doubled the offered load and raised aggregate throughput by 12–14% (RQ2); because offered load was not held constant, this is consistent with the small workload leaving GPU capacity idle but does not on its own establish a consolidation benefit. A load-matched single-engine control (one engine at higher concurrency) or direct utilization telemetry would be needed to confirm it. What remains unknown is whether platform equivalence and the sharing benefit persist under production conditions: larger models (7B–70B), higher concurrency, longer sequences, multi-GPU topologies, and hardware partitioning (MIG or vGPU) with their attendant isolation and quality-of-service guarantees. Future work should extend the evaluation to these regimes, capture direct GPU-utilization telemetry to confirm the under-utilization mechanism, and quantify the configuration-complexity ratings with setup-time and step-count measurements.

Author Contributions: Conceptualization, P.S. (equal) and P.D. (equal); methodology, P.S. (lead) and P.D. (supporting); software, P.S.; validation, P.D. (lead) and P.S. (supporting); formal analysis, P.D. (lead) and P.S. (supporting); investigation, P.S. (equal) and P.D. (equal); resources, M.P. (equal) and H.L. (equal); data curation, P.S. (equal) and P.D. (equal); writing—original draft, P.S. (equal) and P.D. (equal); writing—review and editing, P.D. (lead) and P.S. (supporting); visualization, P.D. (lead) and P.S. (supporting); supervision, M.P. (lead) and P.D. (supporting); project administration, M.P. (lead) and P.D. (supporting); funding acquisition, H.L. (equal), P.D. (equal) and K.G. (equal). All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded in part by the Austrian Research Promotion Agency (FFG) within the RADIUS project, grant number FO999920389, and by the State of Styria (Land Steiermark), Office of the Styrian Provincial Government, Department 12 (Economy, Tourism, Science and Research), within the PRISMA project, grant number ABT12-270413/2024. The article processing charge (APC) was funded by the FFG within the RADIUS project, grant number FO999920389.

Data Availability Statement: The dataset supporting this study is openly available in a Zenodo repository at <https://doi.org/10.5281/zenodo.20667669> (accessed on 25 June 2026). The language model used (KingNish/Qwen2.5-0.5b-Test-ft, revision 7917c64) is available on the Hugging Face Hub.

Acknowledgments: During the preparation of this manuscript, the authors used DeepL (Free) and ChatGPT (GPT-4o, OpenAI) to translate words, phrases, and passages from German into English, and Claude (Opus and Sonnet 4.x, Anthropic) to assist with drafting and revising portions of the text and with writing the Python and TikZ code used to generate the figures and the reported statistical analyses from the study's own benchmark data. No text was published without author review, and no figures, data, or images were AI-generated or fabricated; all figures derive from the authors' own measurements. The authors have reviewed and edited all outputs and take full responsibility for the content of this publication.

Conflicts of Interest: The authors declare no conflicts of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

Abbreviations

The following abbreviations are used in this manuscript:

ANOVA	Analysis of Variance
API	Application Programming Interface
BIOS	Basic Input/Output System
CDI	Container Device Interface
CLI	Command-line Interface
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DMA	Direct Memory Access
GiB	Gibibyte
GPU	Graphics Processing Unit
HPC	High-Performance Computing
IaaS	Infrastructure as a Service
ID	Identifier
IOMMU	Input-Output Memory Management Unit
ITL	Inter-Token Latency
KPI	Key Performance Indicator
KVM	Kernel-based Virtual Machine
LLM	Large Language Model
LXC	Linux Container
MIG	Multi-Instance GPU

MPS	Multi-Process Service
NUMA	Non-Uniform Memory Access
PCI	Peripheral Component Interconnect
PVE	Proxmox Virtual Environment
QEMU	Quick Emulator (unofficial)
RAM	Random Access Memory
SD	Standard Deviation
SM	Streaming Multiprocessor
SMT	Simultaneous Multithreading
SR-IOV	Single Root I/O Virtualization
TOST	Two One-Sided Tests
TPOT	Time per Output Token
TTFT	Time To First Token
USB	Universal Serial Bus
VFIO	Virtual Function I/O
vGPU	Virtual Graphics Processing Unit
VM	Virtual Machine
VMM	Virtual Machine Monitor
VRAM	Video Random Access Memory

References

1. Bhowmik, S. *Cloud Computing*; Cambridge University Press: Cambridge, UK, 2017. <https://doi.org/10.1017/9781316941386>.
2. Cafaro, M.; Aloisio, G. (Eds.). *Grids, Clouds and Virtualization*; Computer Communications and Networks; Springer: London, UK, 2011. <https://doi.org/10.1007/978-0-85729-049-6>.
3. Yang, C.T.; Liu, J.C.; Wang, H.Y.; Hsu, C.H. Implementation of GPU virtualization using PCI pass-through mechanism. *J. Supercomput.* **2014**, *68*, 183–213. <https://doi.org/10.1007/s11227-013-1034-4>.
4. Le, D.N.; Kumar, R.; Nguyen, G.N.; Chatterjee, J.M. *Cloud Computing and Virtualization*; Wiley: Hoboken, NJ, USA, 2018. <https://doi.org/10.1002/9781119488149>.
5. Jin, H.; Ibrahim, S.; Bell, T.; Qi, L.; Cao, H.; Wu, S.; Shi, X. Tools and Technologies for Building Clouds. In *Cloud Computing: Principles, Systems and Applications*; Gillam, L., Ed.; Springer: London, UK, 2010; pp. 3–20. https://doi.org/10.1007/978-1-84996-241-4_1.
6. Yeh, C.Y.; Kao, C.Y.; Hung, W.S.; Lin, C.C.; Liu, P.; Wu, J.J.; Liu, K.C. GPU Virtualization Support in Cloud System. In *Grid and Pervasive Computing*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2013; pp. 423–432. https://doi.org/10.1007/978-3-642-38027-3_45.
7. Menshaw, A.; Fahmy, M. *LLMs in Enterprise*; Packt Publishing: Birmingham, UK, 2025.
8. Walters, J.P.; Younge, A.J.; Kang, D.I.; Yao, K.T.; Kang, M.; Crago, S.P.; Fox, G.C. GPU Passthrough Performance: A Comparison of KVM, Xen, VMWare ESXi, and LXC for CUDA and OpenCL Applications. In *Proceedings of the 2014 IEEE International Conference on Cloud Computing*; IEEE Computer Society: Washington, DC, USA, 2014; pp. 636–643. <https://doi.org/10.1109/CLOUD.2014.90>.
9. Shea, R.; Liu, J. On GPU Pass-Through Performance for Cloud Gaming: Experiments and Analysis. In *Proceedings of the 2013 12th Annual Workshop on Network and Systems Support for Games (NetGames)*; IEEE: Denver, CO, USA, 2013; pp. 1–6. <https://doi.org/10.1109/NetGames.2013.6820614>.
10. Bentele, M.; von Suchodoletz, D.; Messner, M.; Rettberg, S. Towards a GPU-Accelerated Open Source VDI for OpenStack. In *Cloud Computing: 11th EAI International Conference, CloudComp 2021, Virtual Event, December 9–10, 2021, Proceedings*; Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering; Dai, H., Khosravi, M.R., He, Q., Eds.; Springer International Publishing: Cham, Switzerland, 2022; Volume 430, pp. 149–164. https://doi.org/10.1007/978-3-030-99191-3_12.
11. Tian, K.; Dong, Y.; Cowperthwaite, D. A Full GPU Virtualization Solution with Mediated Pass-Through. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC 14)*; USENIX Association: Berkeley, CA, USA, 2014; pp. 121–132.
12. NVIDIA Corporation. NVIDIA Virtual GPU (vGPU) Software Documentation. 2025. Available online: <https://docs.nvidia.com/grid/> (accessed on 18 June 2026).
13. NVIDIA Corporation. NVIDIA Multi-Instance GPU (MIG) User Guide. 2024. Available online: <https://docs.nvidia.com/datacenter/tesla/mig-user-guide/> (accessed on 29 October 2025).
14. Kang, J.; Lim, J.; Yu, H. Partial migration technique for GPGPU tasks to prevent GPU memory starvation in RPC-based GPU virtualization. *Softw. Pract. Exp.* **2020**, *50*, 948–972. <https://doi.org/10.1002/spe.2801>.

15. Eiling, N.; Baude, J.; Lankes, S.; Monti, A. Cricket: A virtualization layer for distributed execution of CUDA applications with checkpoint/restart support. *Concurr. Comput. Pract. Exp.* **2021**, *34*, e6474. <https://doi.org/10.1002/cpe.6474>.
16. Patrão, L. *VMware vSphere Essentials: A Practical Approach to vSphere Deployment and Management*; Apress: Berkeley, CA, USA, 2024. <https://doi.org/10.1007/979-8-8688-0208-9>.
17. Project ACRN. Device Passthrough. 2024. Available online: <https://projectacrn.github.io/latest/developer-guides/hld/hv-dev-passthrough.html> (accessed on 22 October 2025).
18. NVIDIA Corporation. Installing the NVIDIA Container Toolkit. 2025. Available online: <https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/latest/install-guide.html> (accessed on 17 June 2026).
19. Proxmox Wiki. Proxmox VE Wiki Main Page. 2025. Available online: https://pve.proxmox.com/wiki/Main_Page (accessed on 6 October 2025).
20. Proxmox Server Solutions GmbH. *Proxmox VE Administration Guide Release 6*; Lulu Press: Morrisville, NC, USA, 2019.
21. Kärger, T.; Beitter, T.; Nähring, A.; Steil, A.; Zielenski, S. *IaaS mit OpenStack: Cloud Computing in der Praxis*, 1st ed.; dpunkt.verlag: Heidelberg, Germany, 2014.
22. OpenStack Foundation. OpenStack Software. 2025. Available online: <https://www.openstack.org/software/> (accessed on 24 September 2025).
23. OpenStack Nova Project. Hypervisors. 2025. Available online: <https://docs.openstack.org/nova/latest/admin/configuration/hypervisors.html> (accessed on 17 June 2026).
24. Arrichiello, A.; Salinetti, G.; Baude, B. *Podman for DevOps: Containerization Reimagined with Podman and Its Companion Tools*; Packt Publishing: Birmingham, UK, 2022.
25. Podman Contributors. Podman: A Tool for Managing OCI Containers and Pods. 2025. Available online: <https://github.com/containers/podman> (accessed on 15 October 2025).
26. Younge, A.J.; Walters, J.P.; Crago, S.; Fox, G.C. Evaluating GPU Passthrough in Xen for High Performance Cloud Computing. In *Proceedings of the 2014 IEEE International Parallel & Distributed Processing Symposium Workshops, IEEE*; IEEE Computer Society: Washington, DC, USA, 2014; pp. 852–859. <https://doi.org/10.1109/IPDPSW.2014.97>.
27. Jo, H.S.; Lee, M.H.; Choi, D.H. GPU Virtualization using PCI Direct Pass-Through. In *Proceedings of the Information, Communication and Engineering*; Trans Tech Publications Ltd.: Wollerau, Switzerland, 2013; Volume 311, pp. 15–19. <https://doi.org/10.4028/www.scientific.net/AMM.311.15>.
28. Zhou, L.; Song, J.; Yuan, J.; Han, H. Research and Application of GPU Pass-through Based on KVM in Desktop Cloud. In *Proceedings of the 2017 2nd International Conference on Machinery, Electronics and Control Simulation (MECS 2017)*; Atlantis Press: Dordrecht, The Netherlands, 2017. <https://doi.org/10.2991/meecs-17.2017.73>.
29. NVIDIA Corporation. Multi-Process Service (MPS). 2024. Available online: <https://docs.nvidia.com/deploy/mps/> (accessed on 29 October 2025).
30. Yu, P.; Chowdhury, M. Salus: Fine-Grained GPU Sharing Primitives for Deep Learning Applications. In *Proceedings of the Machine Learning and Systems (MLSys)*; MLSys: Austin, TX, USA, 2020; Volume 2, pp. 98–111. <https://doi.org/10.48550/arXiv.1902.04610>.
31. Dhakal, A.; Kulkarni, S.G.; Ramakrishnan, K.K. GSLICE: Controlled Spatial Sharing of GPUs for a Scalable Inference Platform. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC '20)*, New York, NY, USA, 19–21 October 2020; pp. 492–506. <https://doi.org/10.1145/3419111.3421284>.
32. Xiao, W.; Ren, S.; Li, Y.; Zhang, Y.; Hou, P.; Li, Z.; Feng, Y.; Lin, W.; Jia, Y. AntMan: Dynamic Scaling on GPU Clusters for Deep Learning. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*; USENIX Association: Berkeley, CA, USA, 2020; pp. 533–548.
33. Liu, Z.; Chen, C.; Li, J.; Cheng, Y.; Kou, Y.; Zhang, D. KubFBS: A fine-grained and balance-aware scheduling system for deep learning tasks based on Kubernetes. *Concurr. Comput. Pract. Exp.* **2022**, *34*, e6836. <https://doi.org/10.1002/cpe.6836>.
34. Kwon, W.; Li, Z.; Zhuang, S.; Sheng, Y.; Zheng, L.; Yu, C.H.; Gonzalez, J.E.; Zhang, H.; Stoica, I. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP '23)*, Koblenz, Germany, 23–26 October 2023; pp. 611–626. <https://doi.org/10.1145/3600006.3613165>.
35. Agrawal, A.; Kedia, N.; Panwar, A.; Mohan, J.; Kwatra, N.; Gulavani, B.S.; Tumanov, A.; Ramjee, R. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, Santa Clara, CA, USA, 10–12 July 2024; pp. 117–134.
36. Zhong, Y.; Liu, S.; Chen, J.; Hu, J.; Zhu, Y.; Liu, X.; Jin, X.; Zhang, H. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, Santa Clara, CA, USA, 10–12 July 2024; pp. 193–210.
37. Gu, J.; Zhu, Y.; Wang, P.; Chadha, M.; Gerndt, M. FaST-GShare: Enabling Efficient Spatio-Temporal GPU Sharing in Serverless Computing for Deep Learning Inference. In *Proceedings of the 52nd International Conference on Parallel Processing (ICPP '23)*, New York, NY, USA, 7–10 August 2023; pp. 635–644. <https://doi.org/10.1145/3605573.3605638>.

38. Sun, B.; Huang, Z.; Zhao, H.; Xiao, W.; Zhang, X.; Li, Y.; Lin, W. Llumnix: Dynamic Scheduling for Large Language Model Serving. In Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24), Santa Clara, CA, USA, 10–12 July 2024. <https://doi.org/10.48550/arXiv.2406.03243>.
39. Zheng, L.; Yin, L.; Xie, Z.; Sun, C.; Huang, J.; Yu, C.H.; Cao, S.; Kozyrakis, C.; Stoica, I.; Gonzalez, J.E.; et al. SGLang: Efficient Execution of Structured Language Model Programs. In Proceedings of the Advances in Neural Information Processing Systems 37 (NeurIPS 2024), Vancouver, BC, Canada, 10–15 December 2024. <https://doi.org/10.48550/arXiv.2312.07104>.
40. Qin, R.; Li, Z.; He, W.; Zhang, M.; Wu, Y.; Zheng, W.; Xu, X. Mooncake: A KVCache-centric Disaggregated Architecture for LLM Serving. In Proceedings of the 23rd USENIX Conference on File and Storage Technologies (FAST 25), Santa Clara, CA, USA, 25–27 February 2025. <https://doi.org/10.48550/arXiv.2407.00079>.
41. CNCF Container Device Interface Project. Container Device Interface (CDI) Specification. 2024. Available online: <https://github.com/cncf-tags/container-device-interface> (accessed on 17 June 2026).
42. Schuirmann, D.J. A Comparison of the Two One-Sided Tests Procedure and the Power Approach for Assessing the Equivalence of Average Bioavailability. *J. Pharmacokinet. Biopharm.* **1987**, *15*, 657–680. <https://doi.org/10.1007/BF01068419>.
43. Lakens, D. Equivalence Tests: A Practical Primer for *t* Tests, Correlations, and Meta-Analyses. *Soc. Psychol. Personal. Sci.* **2017**, *8*, 355–362. <https://doi.org/10.1177/1948550617697177>.
44. NVIDIA Corporation. Time-Slicing GPUs in Kubernetes. 2024. Available online: <https://docs.nvidia.com/datacenter/cloud-native/gpu-operator/latest/gpu-sharing.html> (accessed on 17 June 2026).

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.