Vrije Universiteit Amsterdam

Honours Programme, Research Thesis

You do not need fast NVMes for MVEs PokeSto: A Storage Benchmark for Modifiable Virtual Environments

Author: Gleb Mishchenko (2766204)
g.mishchenko@student.vu.nl

1st daily supervisor:	Krijn Doekemeijer	(VU Amsterdam)
2nd daily supervisor:	Jesse Donkervliet	(VU Amsterdam)
1st supervisor:	Prof. dr. Alexandru Iosup	(VU Amsterdam)

A report submitted in fulfillment of the requirements for the Honours Programme, which is an excellence annotation to the VU Bachelor of Science degree in Computer Science/Artificial Intelligence/Information Sciences

July 16, 2025

Abstract

Due to popularity and strict performance requirements, online games are a workload of interest for the performance engineering community. The gaming industry yields over $192 billion in revenue and engages over 3.2 billion players [23]. Modifiable virtual enviroments (MVEs) are an emerging game sub-genre with persistence functional requirements. Most popular MVE game - Minecraft - is played by over 140 million people monthly [4], and the Metaverse - a prominent modifiable virtual environment application - market is expected to grow up to 507 billion dollars by 2030 [27]. While storage can and does affect gaming performance and is central for MVE's persistent information storage, there yet is no investigation on how MVEs use it and how it can affect MVE performance. There yet is no available MVE storage traces, no information on the correlation between storage system-level performance and MVE users' Quality of Service (QoS). There is, as well, yet no tool available to measure the impact. Current research consensus outlines a need for application-specific benchmarking when considering the correlation between storage system-level performance and the user-level performance [25]. Existing MVE benchmarks [29]; however, they do not focus on storage performance and limit the exploration supporting only a limited set of workloads. This paper covers this gap. We start the paper by studying the system-level storage traces exerted by different MVE user behavior and construct a model aiding in understanding MVE user storage interaction. Based on this model, we construct a benchmark - PokeSto - supporting trace-based workloads and enabling the exploration of storage influence on MVE user QoS. Finally, we use the benchmark to evaluate the correlation between storage performance and user QoS for a popular MVE server implementation - PaperMC. Our results indicate that there is little correlation between the storage system-level performance and experienced user QoS.

Contents

1 Introduction

1.1 Context

The gaming industry is the world's largest entertainment industry - worldwide, games engage over 3.2 billion players [4] and yield over $192 billion in revenue [23]. In this work, we focus on modifiable virtual environments (MVEs) — a game genre with a defining characteristic of persistent and modifiable virtual environments. Players can change almost every part of the world, including players' and world's states, by attacking players, removing blocks, building blocks, interacting with NPCs, etc. These changes are persistently stored. MVE's biggest representative - Minecraft - played by over 140 million people around the globe [4] and the prominent class of MVE application - Metaverse - is predicted to grow to a 507 billion dollar market by 2030 [27].

MVEs are interesting to study due to their large current and anticipated economic and societal impact. In addition to the aforementioned large market for MVEs, MVEs are an application with a high societal impact in spheres such as primary school education or employee training [31]. By 2030, the MVEs are expected to engage over 2.6 billion people and impact our society by enhanced remote learning, working, social interaction, and more. For our research, MVEs are interesting due to their persistence requirement - MVEs are fast-paced games that need to store all the user modifications persistently using a storage I/O. As the importance of MVEs is growing, there is, as well, an increase in quality of service (QoS) requirements. We already know that even for non-persistent games, the storage I/O can impact performance, and the improvements to storage I/O can increase the user-level metrics such as asset loading times [3]. However, despite growing MVE importance and its inherent need for constant interaction with storage I/O, there is still no research into how the storage I/O performance can impact the user-level MVE performance.

1.2 Problem

The key challenge of this project is the identification of the relationship between the storage I/O performance and the user-level performance under various workloads. This relation is not clear, and even for storage-demanding workloads, such as MySQL workloads, a transition from HDD to storage class memory having a 10000 latency difference [5] yields only 7 times performance improvement [22]. The application-specific translation of storage performance to user-level performance is well known in the performance research community. Therefore, it is the application-specific storage benchmarks that are the current state-of-practice in the community [25]. The application-specific storage benchmarks are already largely adopted by the database performance research community with benchmarks such as YCSB [12] or RockDB [10]. We argue that with an increase in societal importance of MVE and their need for storage I/O interaction to sustain a persistent state, there is a case to be made for MVE-specific storage benchmark. MVEs tend to be closed-source applications that employ a large number of undocumented to the general public storage optimizations. MVEs, as well, are sandbox games with a large workload exploration space. Each player's behavior can yield different storage patterns, making the MVE storage exploration increasingly challenging.

Work has already been done to benchmark a limited subset of MVE player workloads and address MVE scalability concerns [29]. However, existing works do not address the interaction between the MVE and the storage I/O and their impact on the user-level performance. Additionally, current works limit the workloads and do not allow the configuration of MVE workloads at sufficient granularity to represent edge-case player behavior or extend the workloads with player behavior that is unique to a particular server.

In this paper, we cover this gap. We begin by investigating the correlation between the MVE player behavior and the storage interactions, outlining the user-storage interaction model. Then, based on the model, we construct PokeSto - an MVE-specific storage benchmark allowing to *(i)* construct trace-based workloads which can be used to replicate infinitely diverse player behavior at a high level of granularity, and *(ii)* emulate a storage device allowing to test the MVE against storage devices with different storage latency and throughput. We publish the benchmark as an open-source artifact on GitHub. Using this benchmark, we evaluate the impact of storage system-level performance on user-level performance for a popular MVE server implementation - PaperMC. We find that there is little correlation between the system-level storage performance and experienced user performance, with a possibility of running up to **4** MVE instances on the same storage device without any performance degradation.

1.3 Research Questions

In this report, we answer the research questions:

RQ1 What is the relation between user interactions and MVE storage accesses?

MLGs depend on large-scale storage services to operate with good performance. However, it is not clear how these applications interact with the underlying storage systems, and therefore, it is unknown how these storage systems affect MLG performance and other non-functional properties. Modeling these interactions is challenging for three main reasons. First, MLGs are interactive systems whose workload is determined by user activity, which leads to complex workloads that are difficult to capture and reproduce. Second, commercially successful MLGs are typically distributed as closed-source systems, making it difficult to observe their internal behavior. Third, and finally, there are no standardized methods for creating models that link application- and systems-level behavior.

RQ2 How to design and implement a benchmark to evaluate MLG performance based on our storage model?

Because the performance interaction between MLGs and their underlying storage systems are determined by complex workloads and system dynamics, evaluating their performance requires real-world experiments. Performing such experiments efficiently requires a benchmark or performance evaluation framework, but no such system exists. However, designing and implementing such a benchmark is challenging because there is no standardized method for designing such systems.

RQ3 What is MVE scalability on different storage devices?

MLGs are large distributed systems that are typically deployed on cloud computing infrastructure and depend on their underlying storage services. However, because there exists no commonly accepted set of experiments for this purpose, improving our understanding in this area requires designing novel experiments.

1.4 Related Work

The case for application-specific benchmarking In this paper, the authors outline the necessity of application-specific benchmarking as a result of non-linear application-specific correlation between the storage system-level performance and user-level performance. [25]

Yardstick: A benchmark for Minecraft-Like Services In this paper, the authors present a Yardstick - a generalizable benchmark for MVE applications. The authors, as well, use the benchmark to assess the MVE scalability under the increased player count for different server implementations. [29]

1.5 Societal Relevance

This work goes in line with the Dutch Computer Systems Manifesto [18] and addresses the goals of **manageability** and **responsibility** of ICT infrastructure. The ICT infrastructure became essential for a large share of Dutch GDP and is a substantial requirement for many jobs [18]. The dependence of our society on the ICT infrastructure makes the performance and reliability of that infrastructure an important concern for Dutch society. The results of this thesis, including the MVE user-storage interaction model, benchmark PokeSto, and the evaluation of PaperMC, a popular MVE server implementation [7], improve understanding of how applications use the ICT infrastructure and provide the tooling that can be used to determine optimal ICT configurations to host these applications.

1.6 Research Relevance

The results of this thesis improve the understanding of how a popular application, MVEs, uses the storage devices under different user interactions. This thesis, as well, provides a tool for further investigations into the correlation between storage system-level performance and user-level performance. Finally, in this thesis, we evaluate the correlation between the storage performance and user-level performance for a popular MVE server implementation - PaperMC [7] - expanding the knowledge on the types of systems that are optimal to use for hosting this application.

1.7 Plagiarism Declaration

This thesis work is my own work, and has not been copied from any other source (person, Internet, or machine), and has not been submitted elsewhere for assessment.

Table 1: Summary on included user interactions.

User interaction	Read/Write	Data interacted with
Server join	Read	Terrain, NPC, and player data
Terrain modification (block placement or removal)	Write	Terrain data
Server leave	Write	Terrain, NPC, and player data
Position change	Read / Write	Terrain data, NPC data
Player attack	Write	Player data
Player killing NPC	Write	NPC data

2 System Model

In this section, we answer **RQ1** by studying MVE user storage interaction and summing it up into the MVE user-storage interaction model. We begin the section by studying MVE storage accesses patterns observed when executing a subset of user interactions in Section 2.1. Then, based on the observed I/O patterns, we identify several MVE storage I/O routines and construct a generative model in Section 2.2. Finally, based on the generative model, we construct a full user-storage interaction model in Section 2.3. This model, answering an **RQ1**, serves as a road-map to both MVE developers and performance engineers on the persistent data handling techniques employed by MVEs.

2.1 Studying MVE storage accesses

We analyze MVE storage accesses by recording I/O patterns triggered by a subset of user interactions. The studied subset, chosen for convenience, is summarized in table 1. While a particular MVE can have a wider set of user interactions available or some interactions listed in the section, unavailable, we consider the list of observed I/O patterns as comprehensive. The selected set of user interactions yields I/O patterns targeting both reads and writes of persistent data of all the types we considered for this section - **terrain**, **NPC**, and **player data**. To execute user interactions, we used the **mineflayer** library [11]. To trace storage accesses, we use **strace** [28] - a Linux-based tool for trapping system calls. We traces all related file-system calls - **openat**, **close**, **write**, **read**, **pread64**, etc. As an implementation reference, we choose PaperMC - a state-of-the-practice, high-performing MVE server implementation [7]. We are interested, in particular, in server as it where the game state, shared among connected players, is stored.

As a result, we found out that there is a correlation between the total number of I/O operations and the number of unique regions used in workload execution. Storage accesses are random and have a small request size not exceeding **8 KiBs**, which is dictated by the structure of MVE region files. Accesses can be subdivided into instantaneous and delayed depending on the time of execution with respect to the execution time of the triggering user interaction. If the interaction is delayed, its execution is dictated in accordance with some policy that dictates both the synchronization interval and the maximum IOPS during the synchronization.

Observation 1. PaperMC vertically separates different data types using different directories for each. We start by analyzing the structure of the data stored as a result of PaperMC execution. PaperMC uses file-based storage, which is the reason in later sections we trace storage accesses on the file system level. Using the information about file structure, in the later sections, we reverse engineer the purpose of individual file system requests and explain them in terms of user interactions that trigger them. We summarize the observed directory structure in fig. 1. From the diagram, we exclude the Minecraft-specific division of terrain into dimensions, and exclude Minecraft-specific **points of interest** data, which saves the information about places of interest such as ender portals or villages, on the Minecraft map. Every data type - **terrain**, **NPC**, and **player** data - is stored in a specific folder. Using this properly, later, by analyzing the path of executed file system operations, we distinguish write and read operations by accessed data type. Player data is stored on a per-player basis within a corresponding file. Terrain and NPC data, on the other hand, is stored on a per-region basis, with every file further subdivided into sub-regions [1]. Every terrain and NPC subregion contains block data and NPC data, respectively. Dividing region data into smaller sub-regions can help to subdivide the map into smaller units, minimizing the number of bytes that need to be accessed either for data

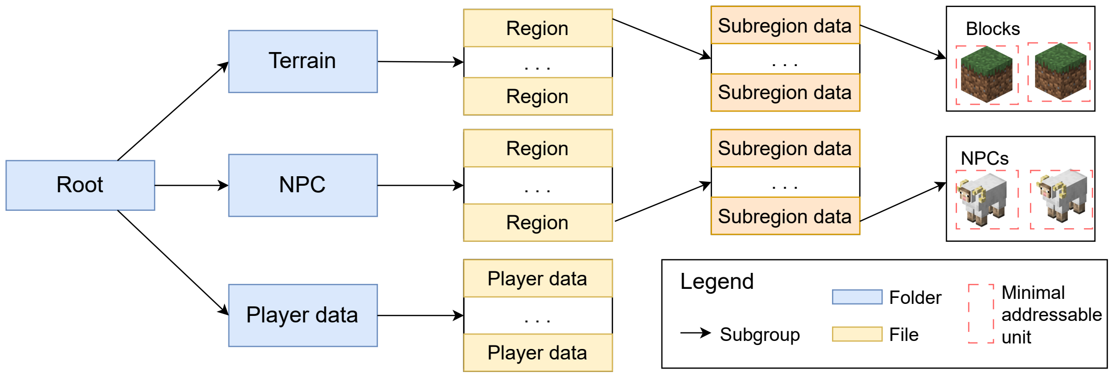


Figure 1: PaperMC directory structure.

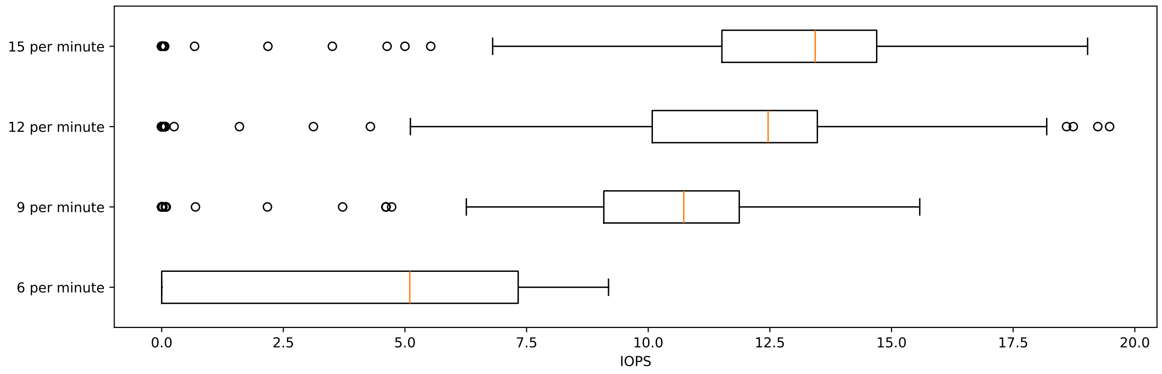


Figure 2: IOPS from teleportation

update or retrieval. The drawback, however, is the increase in the number of I/O operations. The decision of subregion size, therefore, is a trade-off for a particular MVE between the number of bytes accessed and the number of issued I/O operations.

Observation 2: The number of file system accesses increases with the number of requested regions. As terrain and NPC data are organized on a per-region basis, we tested how the number of file system accesses scales when different numbers of regions are required to execute the workload. To trigger a workload that requests a specified number of regions, we executed join and teleportation on a non-pregenerated instance. The join workload is executed with a different number of players spawning in unique regions, with every player spawning in the center of a region. Thus, spawned players act as a proxy for the number of regions players spawn in. Each player is spawned in the middle of the region, and therefore, 1 region is loaded per player. We determine that whenever this interaction is executed, PaperMC has to read both terrain and NPC data for every new region players spawn at. To test teleportation interaction, we teleport a varying number of players to the center of unique regions. In this case, player count acts as a proxy for a number of regions where players are teleported to. Due to the buffering, described further in this section, we teleport each player twice and report only results obtained at the player's second teleportation. We determine that after generating terrain and NPC data, PaperMC will, at a pre-defined interval or whenever the player moves out of the region, write the data back to storage. By testing both interactions, we record how the number of file system accesses scales for interactions that trigger reads and writes. Our data indicates that the number of file system accesses scales linearly with the number of regions used in the workload. We show this both teleportation interaction in fig. 2.

Observation 3. PaperMC storage writes can be classified into instantaneous and delayed. We observed instantaneous writes when players teleport and delayed writes when players attack NPCs. Instantaneous writes are the writes that are executed immediately after the player interaction.

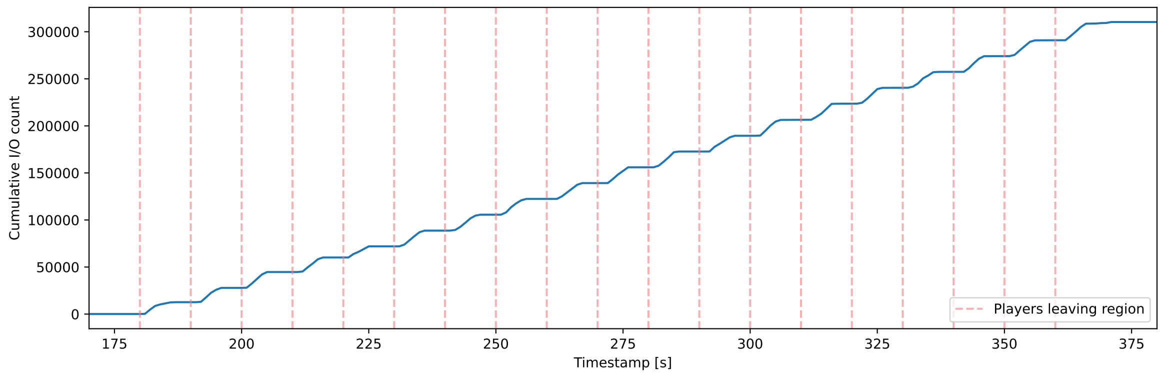


Figure 3: Teleportation event-based flushing policy.

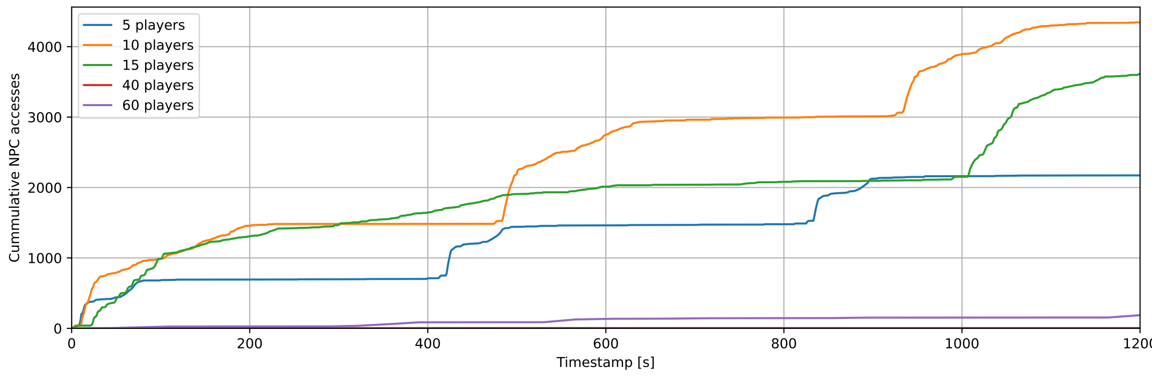
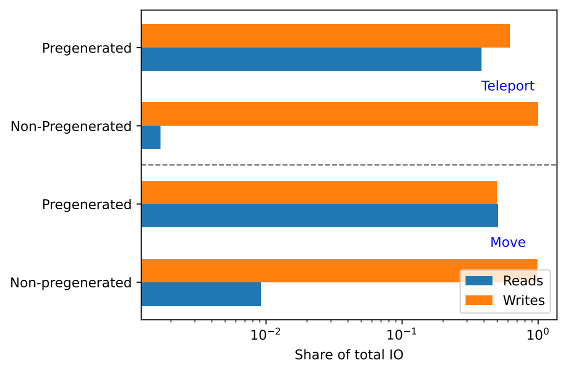


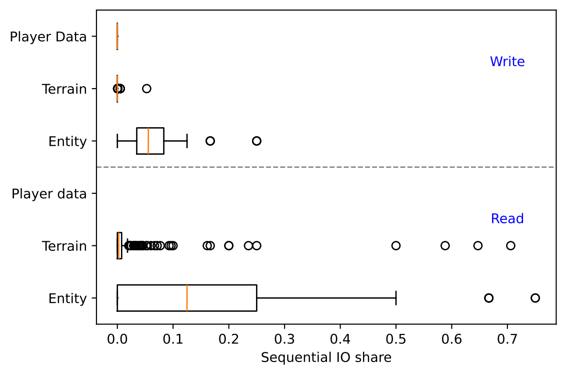
Figure 4: NPC dynamic timer-based flushing policy.

We find that when the region data is flushed to the storage, whenever a player leaves the region. We show this in fig. 3. To record this data, we teleported 10 players in round-robin fashion at a 10-second interval on a non-pre-generated world instance. We can observe that whenever a player leaves the region, there is an increase in the cumulative I/O count. On the other hand, whenever a player hits an NPC, the updates are buffered and flushed at pre-defined intervals. We show this in fig. 4. To obtain this data, we executed player attacking NPCs workload with each player attacking NPCs in their unique region, killing an NPC at 10-second intervals. We can observe the presence of interval-based saves and interval changes depending on the number of players in the workload. The higher the player count, the more frequent the saving interval - while during workload execution **5** and **10** players save their state three time, **60** players' saving interval is too high to observe any saves performed within a given time frame. Executing behavior for longer is not feasible, as an increased number of NPCs required to sustain killing NPCs at pre-defined intervals leads to performance instability, making measurements not possible. We can also observe a decrease in throughput with an increased player count. The increased saving interval and decreased throughput lead to greater inconsistency between data saved in primary and secondary storage, as the data is persistently saved to storage at lower intervals; however, a lower interference between data saving subroutines and server performance is a known problem for vanilla Minecraft [30]. This inconsistency performance trade-off, later in work, we refer to as **policy**.

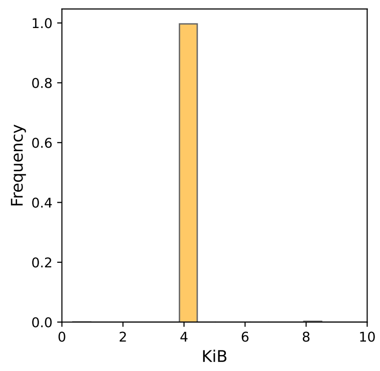
Observation 4. PaperMC has primary random I/O within files, with a majority of request sizes of 4 or 8 KiBs. The R/W ratio varies depending on the world generation state. We find out that all the files opened during PaperMC execution are opened in an **indirect** mode. We note that during MVE gameplay, the R/W ratio always shows a higher number of writes. Even when players are moving on pre-generated instances through both means of simple movement and teleportation, the number of writes exceeds the number of executed reads. We show this in Figure 5a. We show that the majority of terrain and NPC accesses are non-sequential in Figure 5b.



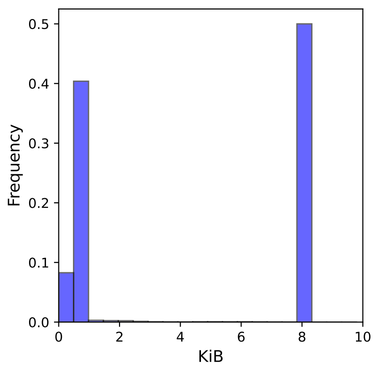
(a) Share read/write per generation state.



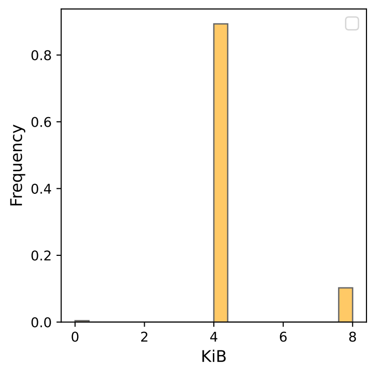
(b) Sequential I/O percentage.



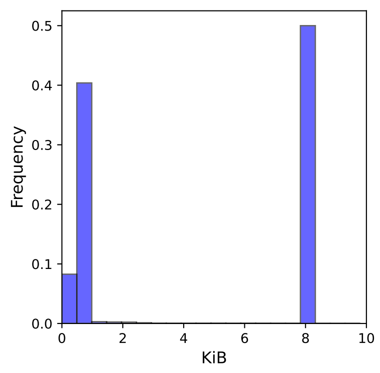
(c) Region reads request size.



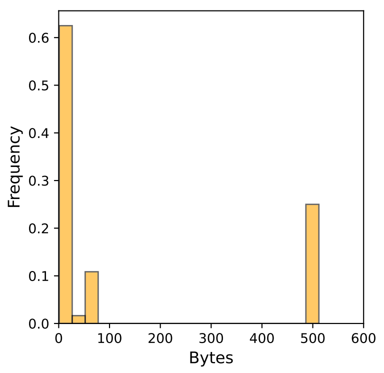
(d) Region writes request size.



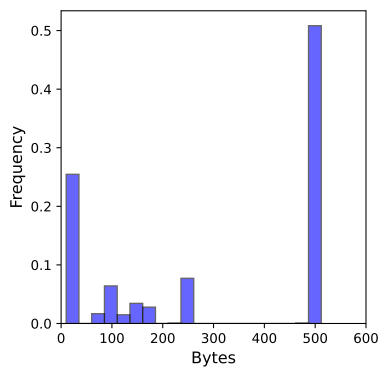
(e) NPC reads request size.



(f) NPC writes request size.



(g) Playerdata reads request size.



(h) Playerdata writes request size.

Figure 5: Storage accesses evaluation.

We define sequentially as the percentage of consecutive writes and reads. We define storage access to be sequential if $pp + ba = cp$ where pp stands for the previous access pointer, ba stands for the number of bytes accessed, and cp stands for the current access pointer. File system randomness arises from PaperMC file structure, dividing region files into subregions (chunks) having a size of multiple of 4KiB and header sections that have pointers to the beginning of different subregions within files [2]. The header size is set to be 8 KiBs and sub-region sizes are multiples of 4 KiBs. This can be observed, as well, in the request size distribution in Figure 5. Whenever either the region or NPC data are read, the reads are either 4 or 8 KiBs - first, the region header is read, and then the required subregions. Writes are executed both in 8 KiBs for header updates and at sizes lower than 4 KiBs for subregion update - the write size depends on the number of structures or entities within the region.

Table 2: Generative model events.

User interaction	Access count	Unique file count	I/D	R/W	Data types
Region load	1 per region + 1 per subregion	1 per region	I	R	Terrain and NPC
Region flush	1 per region + 1 per subregion	1 per region	I	R	Terrain and NPC
NPC update	1 per region + 1 per subregion	1 per region	D	W	NPC
Player update	1 per player	1 per player	I	W	Player data
Player load	1 per player	1 per player	I	R	Player data
Terrain modification	1 per region + 1 per subregion	1 per region	D	W	Terrain data

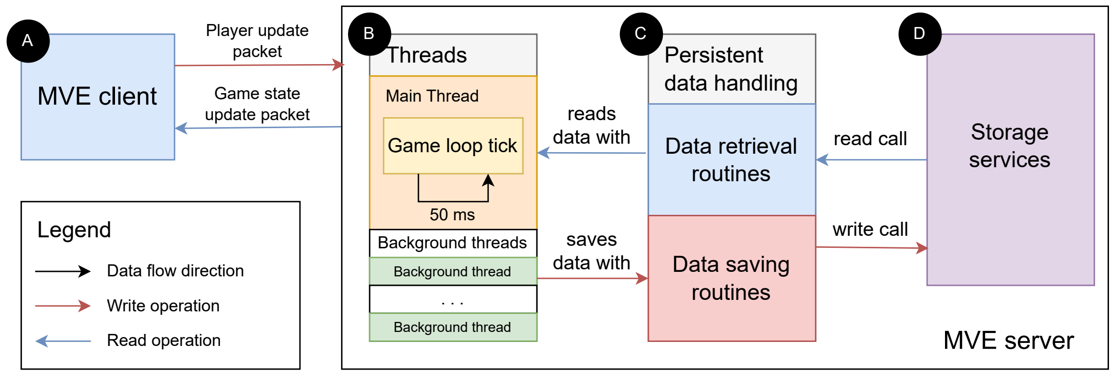


Figure 6: User storage interaction high-level overview.

2.2 Generative model for MVE interactions

Based on the results from Section 2.1, we construct the generative model in Table 2. To make a model generalizable to other MVEs we introduce a notion of **I/O routines**. **I/O routine** is an MVE function that handles the persistent data. We consider this abstraction generalizable to MVE interactions not included in the storage access study in Section 2.1 as implementation of different user interactions are likely to reuse abstracted persistent data handling routines rather than re-implementing different persistent data handling for each user interaction. While there can be more storage handling subroutines, we only focus on the ones that were triggered by studied actions from Section 2.1. We consider our list comprehensive, as listed interactions encompass both reading and writing for all data types. We summarize the I/O subroutines in Section 2.1. There, we also, classify whether they were observed to trigger immediate (I) or delayed (D) storage interactions. This classification is important as immediate I/O subroutines will flush data onto the storage right after user actions, while delayed I/O subroutines will first buffer up the data and flush it in bursts. The final list of low-level I/O subroutines we consider is - **region load**, **region flush**, **NPC update**, **player update**, **player load**, and **terrain modification**. These I/O subroutines are be called when **player is joining or changing their position**, **player is leaving a game or changing their position**, **player is attacking NPCs**, **player is attacking the other player or updating the inventory**, **player joining the game**, or **player placing or removing blocks**, respectively. It is important to note that immediate and delayed classification is only applicable to writes; reads are always executed immediately.

2.3 Modeling MVE user-storage interaction

This section present the user-storage interaction model, which we base on information obtained in Section 2.3 and on our abstraction of I/O routines summarized in Section 2.2. This model answers the **RQ1** and serves as a road-map for understanding MVE user-storage interactions. This is useful for both *(i) MVE developers* and *(ii) performance engineers.*

We first introduce the high-level reference model for user-storage interaction in Figure 6. There we define Ⓐ MVE client - machine accepting inputs from a user and rendering an output on their screen

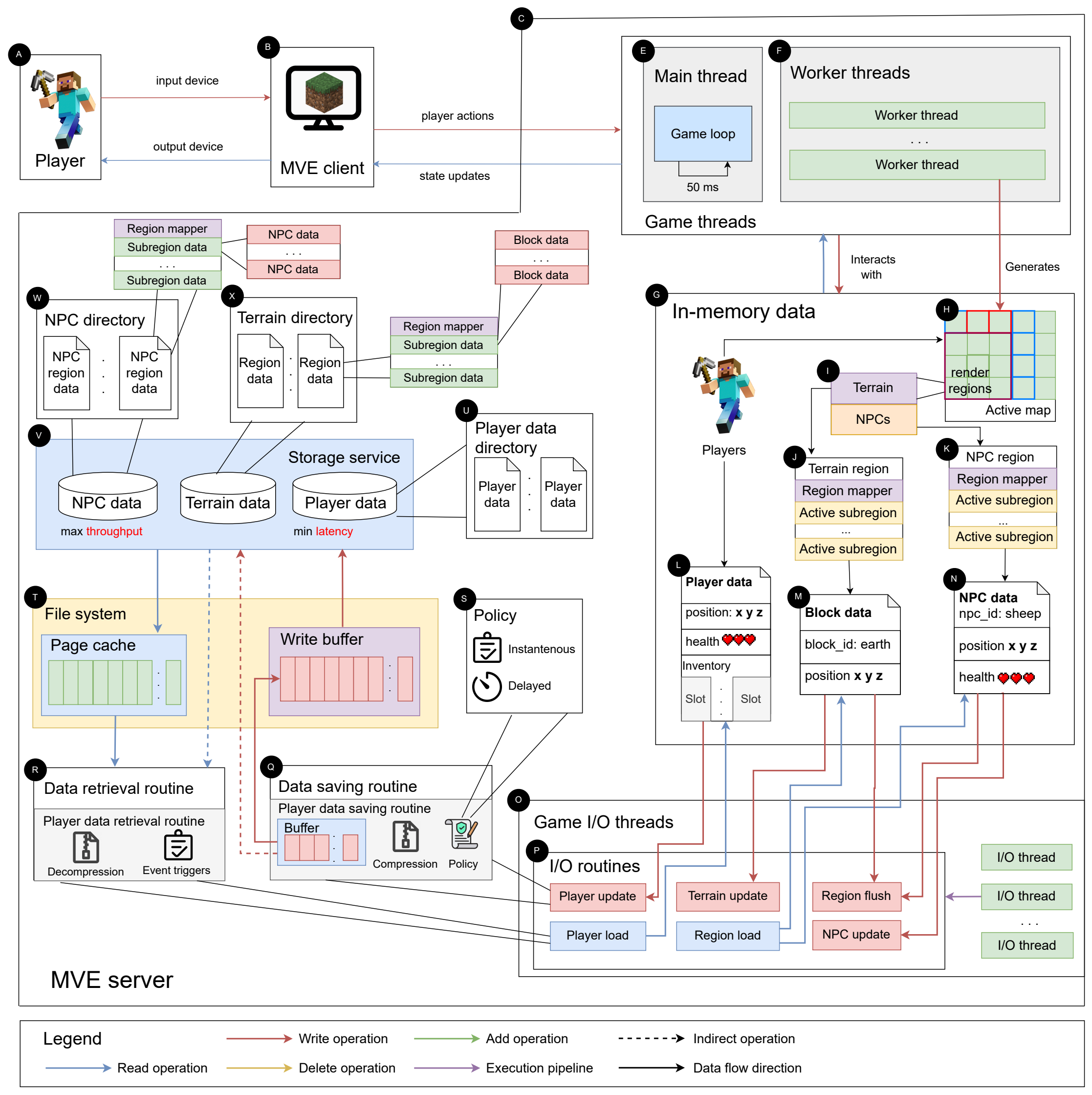


Figure 7: Full user storage interaction model.

- and Ⓑ MVE server - machine accepting requests from multiple clients, processing their inputs, and sending out state updates to render. Within the server, there are a number of Ⓑ threads that are responsible for processing player interactions. Among those of high importance is a main thread with a game loop tick. This game loop ticks at a pre-defined speed, sending out updates to the players - in Minecraft's case, the loop tick interval is 50 ms. The performance of the game loop tick is crucial - if tick time exceeds a pre-defined value, users will experience performance lag. All the blocking tasks, therefore, can be offloaded to a number of background threads. To interact with persistent data, threads use a number of Ⓒ persistent data handling subroutines that, in turn, fetch the persistent data from the Ⓓ storage service.

A more detailed view is summarized in Figure 7. The model starts with a Ⓐ user. A user communicates with a Ⓑ MVE client - an application running on the user's computer - through a means of some input device like a keyboard or a mouse. The client, in turn, renders a game and outputs it through means of some output device(s) like a screen or audio speakers. Whenever a player does any user interaction like walking, placing a block, etc, a client communicates this data to the MVE server. 🅒 MVE server, in turn, periodically sends state updates that the MVE client outputs to the user. The player actions are sent to the 🅔 main thread with a game loop tick running inside. There is also a 🅕 background worker threads that process user actions and generates region data on demand. Processing the user interactions, game threads interact with 🅖 in-memory data. In-memory data stores a representation of 🅗 active map containing information about 🅛 players, 🅜 terrain, and 🅝 NPCs within the active map. We define an active map as a collection of currently rendered regions. For every 🅙 terrain and 🅚 NPC region that is part of the map, the in-memory data contains loaded sub-regions and the file header. The file header is stored to be able to selectively update or fetch sub-regions. Synchronization of in-memory data with persistently stored data and retrieval of persistently stored data into in-memory data is done by a number of 🅞 game I/O threads that call a number of I/O routines. The subroutines that we included in the model are the same ones that we included in the generative model in Section 2.2. The subroutines responsible for reading - 🅡 data retrieval routines - have a number of event triggers, like the player moving into a not-yet-loaded region that triggers the read. If MVE compresses stored data, it can also have a decompression routine. The subroutines responsible for writing - 🅠 - data saving routines have some policy according to which they perform the writes. We classify these policies into instantaneous, triggered by some event, and write to the storage immediately, and delayed, storing updates in a buffer and then flushing them at some interval. If compression is used, a data-saving routine can also have a compression routine attached. Data saving and retrieval routines write to the storage using 🅣 file system using either direct or indirect writes. Indirect reads and writes go through file system performance optimization components - a page cache and write buffer, respectively. Direct reads and writes, in turn, are executed directly on the 🅥 storage service. The storage service contains 🅦 NPC directory with corresponding NPC region files, 🅧 terrain directory with corresponding terrain region file, and player data directory with corresponding player data files. Both NPC and terrain files are subdivided into different subregions and contain a region mapper to dynamically fetch and update subregions. Each terrain and NPC subregion contains information on the blocks and NPCs within the file, respectively.

3 Design and Implementation of PokeSto

In this section, we design and implement PokeSto - MVE scalability with a storage services focus. The benchmark is based on the user-storage interaction model presented in Section 2.3 and of importance to *(i)* MVE server operators in finding our required hardware for desired QoS, *(ii)* infrastructure providers in determining storage QoS requirements, and *(iii)* MVE developers in finding out the required level of application storage requirements. We start the section by posing a number of benchmark requirements in Section 3.1. Then, we offer a high-level overview of benchmark design in Section 3.2. Then, we show the novel elements of design - the trace-based workloads and virtualized storage in Section 3.3 and Section 3.4. Finally, based on the analysis of the yielded storage patterns in Section 2.1 and the constructed storage model in Section 2.2, we design a number of MVE storage workloads in Section 3.7 and costruct a number of relevant system and user-level metrics in Section 3.6.

3.1 Benchmark Requirements

We construct the following benchmark requirements (BR).

BR1 *System compatibility* - as MVEs can be executed on a number of systems, the benchmark has to assume as little as possible on the implementation of the system. Our benchmark design requires only POSIX file support. This is a reasonable assumption as file abstraction is used by the majority of storage systems.

BR2 *Behavior coverage* - MVEs are sandbox games with infinitely many behavior combinations. As different types of MVE servers focus on different game mechanics - for example, a PvP server will focus on player fighting - there is no average MVE user behavior. To be relevant for a full range of MVE servers, benchmark player emulation should support all user interactions presented in Section 2.3.

BR3 *Relevant storage simulation* - allowing the user to benchmark the MVE with a wide range of storage configurations, the benchmark must incorporate a configurable, in terms of throughput and latency of the storage device, storage simulation.

BR4 *Relevant storage and user QoS metrics* - to provide the user with actionable insights based on the benchmark run, the benchmark should capture a relevant set of metrics that reflect both system-level storage performance and user-level QoS.

BR5 *Reproducibly* - to ensure scientific validity, obtained results must be reproducible.

3.2 Design Overview

We summarize PokeSto design that satisfies the benchmark requirements presented in Section 3.1 in Figure 8. The benchmark design is based on the general MVE benchmark - Yardstick - allowing different MVEs to be tested on the basis of the same benchmark architecture. The novelties of our benchmark are **trace-based workloads**, **storage service simulation**, and **behavior generation component**. All benchmark novelties are further discussed in their respective sections. Our benchmark design starts with the 🅐 user that can either supply a 🅒 user-level game trace perfectly reflecting user activity of their server or generate one with the behavior type most appropriate for their use case using the 🅑 behavior generator component. As some user actions depend on the virtual world structure, for example, users can remove only existing blocks, and user can supply their MVE world data. Alternatively, the user can use a 🅝 world setup routing that ensures an identical system under test (**BF5**) during different benchmark runs. Player traces are executed using the 🅓 player emulation component that simulates player actions on top of the running 🅔 MVE instance. MVE instance is running on top of some 🅙 file system, which is, in turn, running on top of a 🅘 simulated storage service. The user can set up this storage service by specifying a static latency and throughput (**BF3**). Both file system and user-level metrics are pulled by a 🅚 metrics collector component that collects both relevant user QoS and file system usage data. The result data is then stored at the pre-defined 🅛 storage location.

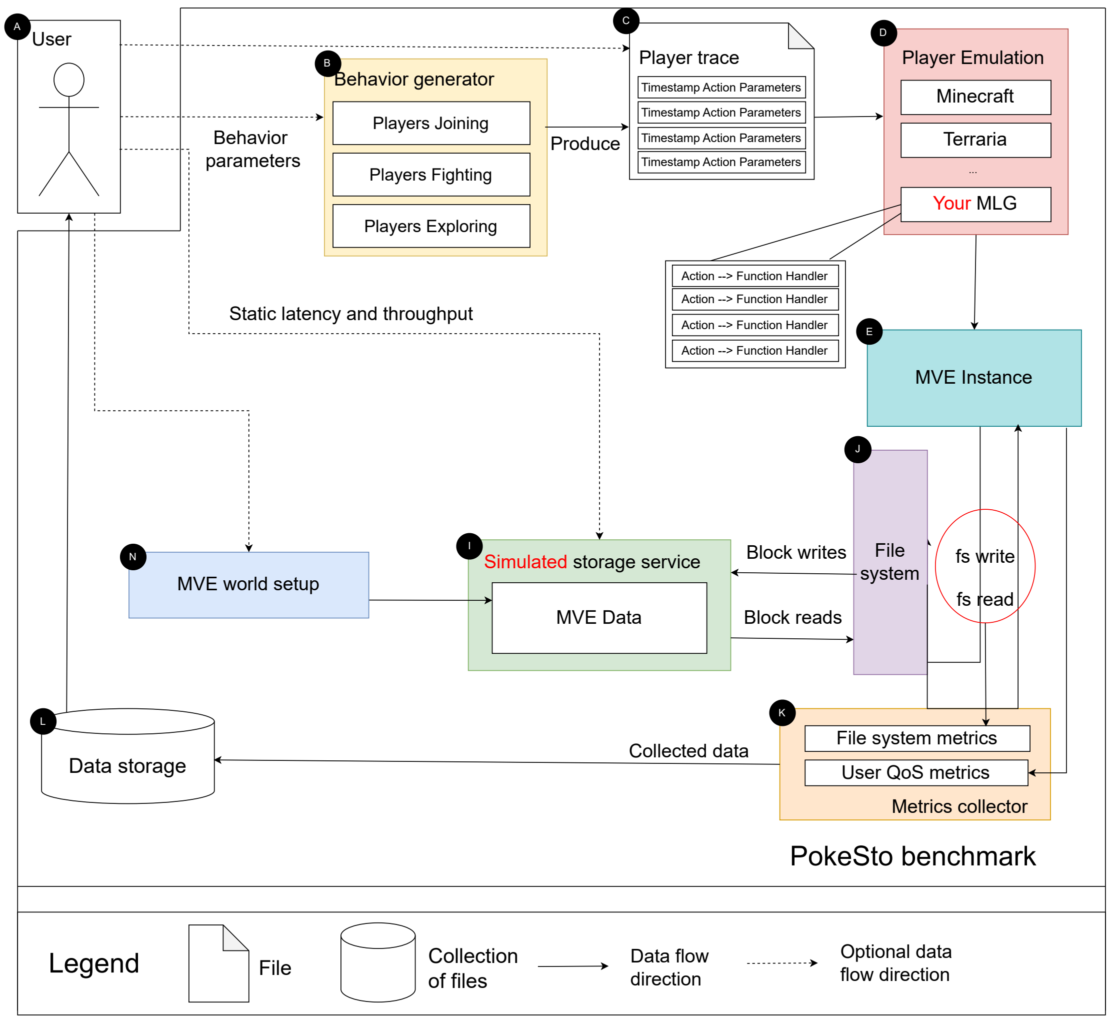


Figure 8: PokeSto design overview.

3.3 Trace-Based Workloads

Our benchmark design is based on trace-based workloads. Those, compared to Yardstick's workloads have an advantage of *(i)* reproducibility (**BR5**), and *(ii)* simulation of arbitrary complex behavior (**BR2**). Experiment reproducibility is achieved by allowing to, as well, supplying persistent world data or supplying world setup traces that ensure world persistent data on top of which the workload is executed, stays the same for different benchmark runs. Representation of arbitrarily complex behavior is useful to the benchmark users, as it can be used to benchmark player behavior relevant to their particular use case. To ensure that arbitrarily complex behavior can be executed in a reproducible manner, we propose a player emulation design summarized in Figure 9.

At each run, the Ⓐ trace file, containing in every line the *(i)* action timestamp, *(ii)* action type, and *(iii)* action parameters, is read by the Ⓑ player emulation component. At the interval of 50 ms - Minecraft tick loop interval - Ⓒ main thread will check if any actions need to be executed. If there are, this action will be passed onto a Ⓓ player thread, which is unique for every emulated player. This player thread executes actions one by one, which is consistent with the way the player's actions are executed in a real-world gameplay scenario. Ensuring actions are run sequentially, every user action is put into the Ⓔ action queue, which is attached to a Ⓕ consumer that consumes actions one by one whenever the previous action has completed. Whenever the action is fetched, it is executed

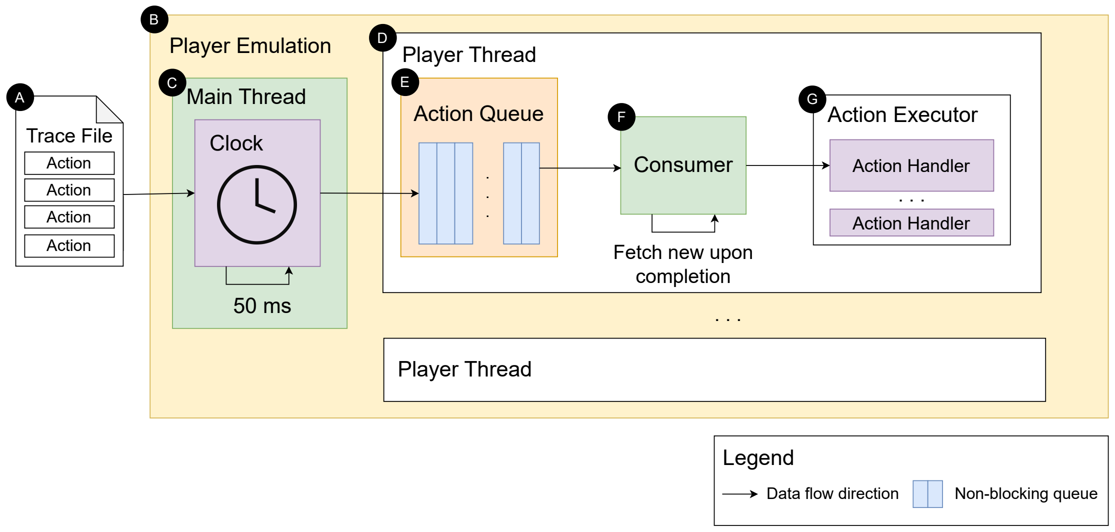


Figure 9: Player Emulation Design Overview

Table 3: Latencies Used to Report Results.

Latency	**Throughput**	**Reference Storage Configuartion**
0 μs	19.2 GB/s	Baseline, ideal memory-like storage configuration [9]
7 μs	10 GB/s	Storage-class Intel-Optane-like memory [20]
50 μs	5 GB/s	NVMe SSD [26]
100 μs	550 MB/s	SATA SSD [16]
1 ms	4 GB/s	io2 Block Express, io2, gp3, gp2, st1 optimistic approximation [8]
6 ms	140-160 MB/s	HDD or io2 Block Express, io2, gp3, gp2, st1 pessimistic approximation [8] [24] [15]

by a corresponding G action handler that notifies the consumer back whenever the action has been completed. For the actions that can fail, the action handlers can, as well, incorporate a timeout after which the action is marked as completed.

3.4 Simulated Storage

To evaluate the effect of storage system-level performance on user-level QoS, the benchmark incorporates a storage simulation with a configurable *(i)* throughput and *(ii)* latency. The structure of the storage component is summarized in Figure 10. There, a user can specify a static throughput and latency of the storage device. Whenever the A file system sends read or write requests, the B block layer will put it in the C request queue from which the requests are dequeued according to the E clock, which controls the latency and throughput emulation. All storage operations are executed on F memory, guaranteeing negligible read and write overheads. Whenever the operation completes, it is put in the D completion queue, where, through the block layer notification, the file system fetches the result of the operation. As both the latency and throughput of storage devices vary greatly for each device model and can change depending on the workload and the storage device load [13], we select, for convenience, a ballpark latency and throughput performance baseline, which are presented in Table 3.

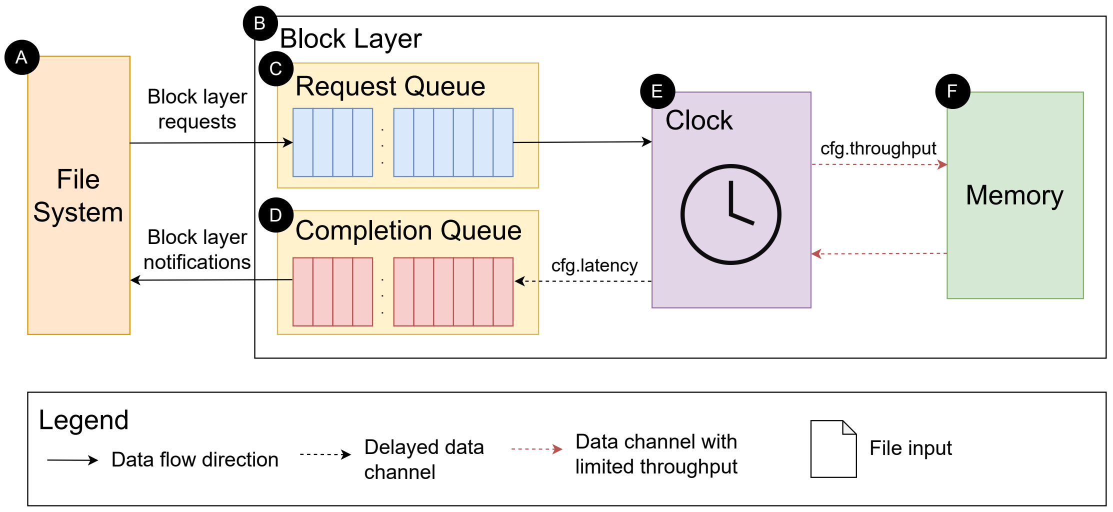


Figure 10: Simulated Storage Overview

Table 4: Behavior list for generation.

Behavior	Action frequency equivalent (per second)	Focus
Instance start and termination	Instance start and terminations	On-demand provisioning
Player exploration	Regions explored	Parkour, elytra flying, any server with infinite world
Joining and leaving	Players joined or left	Server load variability
Player attack	Issued attacks	PvP servers
Player terrain modification	Blocks removed or added	Build, survival servers
NPC attack	NPCs killed	Survival server, NPC farms

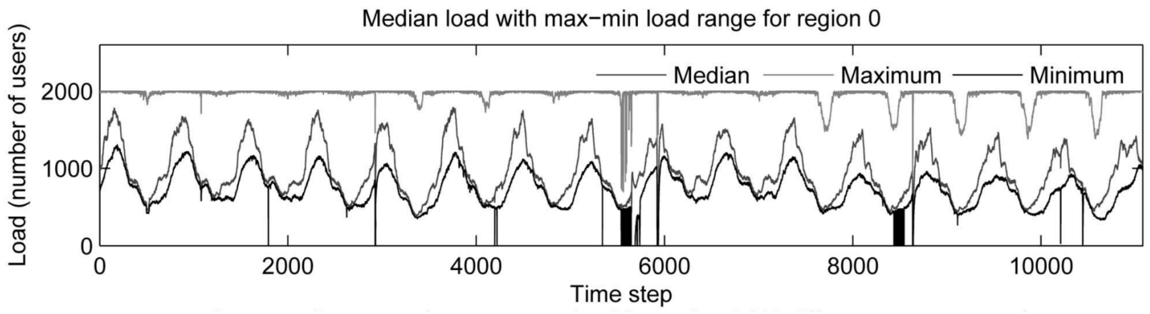
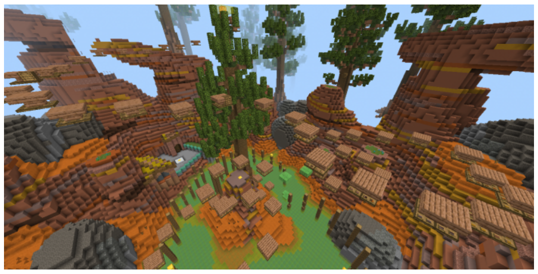


Figure 11: Performance Variability [21]

3.5 Behavior Generation

To allow the user to test the workload in the absence of user activity traces, we construct a list of player behaviors that can be used out of the box. This is beneficial as it allows the stakeholders to test MVE player behavior that is yet to be present on their servers. This can be useful in estimating the required resources to launch a new server type, for example, a PvP server. Each generated behavior is a list of similar actions. For each player behavior, there is some metric of periodicity - a unit that expresses how often the action is executed on the server. For example, for players joining and leaving, this metric is players leaving per second and players joining per second.

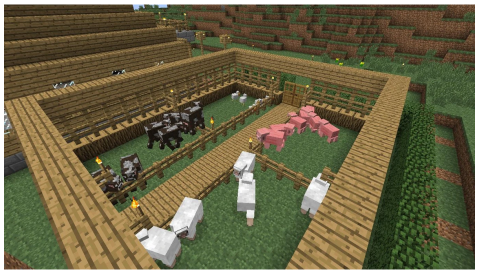
Each behavior included presents a particular interest to some server type or commonly observed



(a) Parkour

(b) Elytra Racing

(c) PvP server fight



(d) Farm

Figure 12: Different servers and game structures.

Table 5: Benchmark Metrics Overview.

Metric	Level	Focus	Method
Number of file system requests	System	Required operation count	Strace
Number of bytes requested	System	Required throughput	Strace
Tick time	Application	Server user-level performance	Telegraf
Completion latency	Application	User perceived latency	Timer

world structures. Player exploration workloads are interesting for any server that has an explorable big world in which players can move. As well, it is of particular interest to parkour [6] or elytra racing servers [14] (shown in Figure 12a and Figure 12b) where players do not modify the terrain but only load already existing one. Player fighting is of particular interest to PvP servers, which are focused solely on fighting behavior. Joining and leaving is particularly interesting to all servers due to observed performance variability Figure 11; however, of particular interest of large-scale servers that can experience large bursts. Both terrain modification and NPC attack are interesting for a survival server, as all *mining*, *building*, and *killing NPCs* are part of the resource retrieval game mechanic. Terrain modification workload is, as well, of particular interest to the *building* servers. Those servers tend to be in creative mode with players focused on solely on building. NPC attack workload is of particular interest to the farm (Figure 12d) - a structure to kill large numbers of NPCs either manually or automatic for a resource gain.

3.6 Metrics

This section defines a list of collected system-level and user-level metrics. System-level metrics reflect the file system usage. That information can be used by *(i)* server operators, *(ii)* infrastructure operators, and *(iii)* further researchers. *(i)* Server operators can use to estimate both a total request count and the requested bytes that are required for their server. This is especially relevant for server operators using pay-as-you-go cloud infrastructure in cost estimation. Knowing the total number of requests and the number of requests *(ii)*, infrastructure operators can use this information to estimate the required number of operations and bytes that their storage infrastructure can support. Further researchers can use the storage traces for MVE emulation and use it for further research into the performance improvement of cloud-based storage services. User-level metrics can be used by stakeholders to estimate the required storage service QoS level for a particular use case. A summary of benchmark

Table 6: Benchmark Workload Overview.

Workload	Triggers	Behaviors
Read event trigger	Data retrieval subroutine	instance start-up, players joining, movement on pre-generated instance
Interval-based save trigger	Data saving subroutine	Position static behaviors such as player fight, NPC attack, or terrain modification.
Event-based save trigger	Data saving subroutine.	instance termination, movement on non-pregenerated instance

metrics is provided in Table 5.

System-level metrics include *(i)* total number of requests and a *(ii)* a total number of requested bytes. Both metrics are recorded using **strace** tool - a Linux-based tool for trapping file system calls. We record file system-level metrics for reproducibility (**BF5**) and system compatibility requirements (**BF1**) - for storage-level metrics, we want the results to be reproducible independent of the implementation of the underlying system. Additionally, recording storage accesses at that level allows for reverse-engineering the semantic meaning of the requested data, ensuring a higher scientific validity of results. For example, semantic meaning can be deduced by accessing file names or directory names. **User-level metrics** include the *(i)* tick time and *(ii)* completion latency. Tick time is recorded via *Telegraf* [17] - a tool that we use to measure Minecraft performance, and completion latency is measured by a UNIX get time of the day timer. We distinguish between these two metrics as while *(i)* shows the performance that will be perceived by all the players independently of the user(s) that execute an interaction and *(ii)* task completion time is observed only by the user(s) that execute an interaction.

3.7 Workloads

We base executed workloads on the model presented in Section 2. We want our set of workloads to cover all the points of application interaction with the storage services. Those are *(i)* instantaneous or *(ii)* delayed data saving I/O routines or *(iii)* data retrieval I/O routines. Shown in Section 2.1, every workload can be triggered via some user interactions. Thus, every workload defined in PokeSto must execute a behavior that leads to the type of interaction prescribed by its focus. The list of workloads, their focus, and associated behaviors is defined in Table 6. Using the benchmark, users can test behaviors outside of the executed workload scope; however, using the proposed workloads suggests the possible performance impact. For example, workload focusing on *(i)* interval-based saving can be used to show performance degradation at the time of saves at pre-defined intervals, while *(ii)* event-based saving and *(iii)* can show performance degradation at the time of event execution. Every workload is defined through a: *(i)* number of players, *(ii)* player count, *(iii)* action frequency per player, *(iv)* region count. Both *(i)* and *(iii)* act as proxies for the total number of actions that are executed during a time interval. There needs to be *(i)* present as there is likely to be some limit for the number of actions a player can execute at a time. As well presence of *(i)* is important as it can change the number of accesses issued to update or retrieve players state. *(iii)* Number of regions as the MVE map is organized into separate regions, and increasing the region count will lead to an increase in the number of accesses and number of unique files. It is important to also note that for the region to become active, there needs to be at least one player present. Thus, at all times *(iii)* <= *(i)* must hold.

Table 7: Experiment Overview.

Focus	IV	DV	Interest group	Section
Effect of storage latency and throughput on MVE scalability	Active region count	User QoS	MVE operators	Section 4.3
Effect of storage collocation on MVE scalability	Instance count	User QoS	Cloud infrastructure operators	Section 4.4

4 Evaluation

In this section, we evaluate the scalability of the popular MVE server implementation - PaperMC - using a PokeSto designed and implemented in Section 3. We start this section by defining the experiment setup in Section 4.1. Then, we summarize the main findings and provide MVE infrastructure operators with a number of insights in Section 4.2. Later, in Section 4.3 and Section 4.4, we analyze the single and multiple instance scalability.

4.1 Experiment Setup

We designed our evaluation to answer **RQ3**, primarily focusing on MVE scalability, focusing both on single instance scalability and multi-instance scalability where multiple PaperMC instances are collocated on the same storage drive. We summarize the experiments and their focus in Table 7. To address the research question, we begin the investigation by assessing the relationship between the storage medium used and the MVE instance scalability. We asses both the average use case and stress-test use-cases. We define stress-test use-cases as ones yielding the highest number of storage accesses. We use the number of regions used in the workload as a proxy for instance scalability. We consider this as a good proxy to asses the MVE scalability on a particular storage device since, in Section 2.1, we show that its increase increases both the number of file accesses and the number of unique files accessed. Having information on single instance scalability, we move on to multi-instance setup, where we investigate the impact of MVE collocations on the same storage drive. In this case, our proxy for scalability is the number of collocated instances. The focus of experiments in both cases is the translation of storage-level performance into user-level performance, and in both cases, user QoS, measured in metrics defined in Section 3.6, tick time, and task completion latency, is a dependent experiment variable.

To conduct an exhaustive investigation of single instance scalability, we test a popular MVE server implementation - PaperMC [7] - against all three workloads described in Section 3.7 - **read-trigger**, **instantaneous write-trigger**, **delayed write-trigger**. For each workload, we select the player behavior that yields *(i)* storage behavior prescribed by a workload, *(ii)* found to be the most storage-intensive, and *(iii)* representative of MVE player behavior. For **read-trigger**, that is players joining; for **instanteneous write-trigger** it is players teleporting on non-pregenerated instances, and for **delayed write-trigger** it is players attacking NPCs. To emulate player behavior, we use benchmark's trace-based workloads (Section 3.3). To execute player interactions, the component uses the **mineflayer** [11] library. We measure experiment results using the metrics collector component introduced in Section 3.2, recording both **system-level** and **user-level** metrics. **User-level** metrics are recorded using **Telegraph** for tick time and a get time of the day **UNIX timer** for task completion time. **System-level** metrics are captured by **strace** - Linux-based file system tracing tool [28].

For consistency reasons, we use the same setup throughout all experiments. The storage is emulated using **nullblock** tool [19] - a Linux-based block device emulation tool. Used storage latencies and throughputs are summarized in Section 3.4. Every experiment is executed using PaperMC **1.8.1** - the latest version at the time of the beginning of the project. PaperMC ran on top of the **ext4** file system - a default Linux file system. The Linux kernel version used is **5.12.0+** and the Ubuntu version used is **20.04 LTS**. For hardware, we used an available research compute cluster with **Intel(R) Xeon(R) Silver 4210R** CPU with TurboBoost disabled and **256 GB** RAM. All the experiments were then run in a QEMU virtualized space with **64 GB** of RAM and **16** CPU cores allocated. Each Minecraft instance was allocated **32 GB** of RAM to ensure that RAM is not a bottleneck. To ensure that the system is not bottlenecked by terrain generation, we allocate **12** worker threads - considerably

Table 8: System under test overview.

Component	Value
VM	QEMU with 16 cores and enables hardware acceleration
CPU	Intel(R) Xeon(R) Silver 4210R CPU with TurboBoost disabled
Core count	64
RAM	32 GB
Linux Version	5.12.0+
Ubuntu Version	20.04 LTS
File System (VM)	Ext4
Version	PaperMC v1.8.1
Worker threads count	12
World state	Flat
World seed	42

higher thread count than recommended, even for the maximum number of players participating in our experiments.

4.2 Main Findings

This section acts as a summary of the main findings and insights for both single-instance [SI] and multi-instance [MI] scalability experiments (Section 4.3 and Section 4.4). The set of main findings is:

MF1 (SI) In our setup, a faster storage device can improve the lag tick occurring as a result of more than 160 regions being loaded per second. [up to 3.5 improvement from HDD and 2.7 from io2 from memory-like storage baseline] (Section 4.3).

MF2 (SI) In our setup, for any other workloads and metrics measured, there is **no difference** between using **HDD** and ideal-case **memory-like** storage device (Section 4.3).

MF3 (MI) In our setup, an HDD with a single request/response queues can handle at least up to **four** PaperMC servers running in parallel (Section 4.4)

The set of main insights is:

MI1 (SI) Unless a server expects large join bursts, PaperMC operators can afford to go for **lower QoS** storage (Section 4.3).

MI2 (MI) A server operator can collocate multiple PaperMC servers on a single storage drive without any loss in performance (Section 4.4)

4.3 Single Instance Scalability

This section evaluates MVE single instance scalability for different storage devices, focusing on user QoS measured in application-level metrics defined in Section 3.6. As defined in Section 4.1, this section both considers average and stress-testing use-cases. As a result of this section, we find that storage has almost no effect on PaperMC scalability, both for average and edge cases. The only scalability issues that can be improved with faster storage are tick time lag occurring as a result of a large number of reads. For every other experiment, there is no correlation between the used storage and observed MVE scalability. As we test both average and the most storage-intensive behavior, we extrapolate that for other player behaviors, storage will also not influence MVE scalability.

Observation 1. Under Yardstick-defined WalkAround [29] workload, storage does not affect user-level performance. Under the Yardstick-defined baseline workload, there is no difference between running PaperMC using memory-like storage or HDD. To obtain this data, we run the default Yardstick WalkAround workload [29] with 10 players spawning in the center of the map. We show the tick time when the PaperMC is run on top of HDD and memory-like storage in Figure 13. We find that there is no significant correlation between the storage device and the experienced user-level

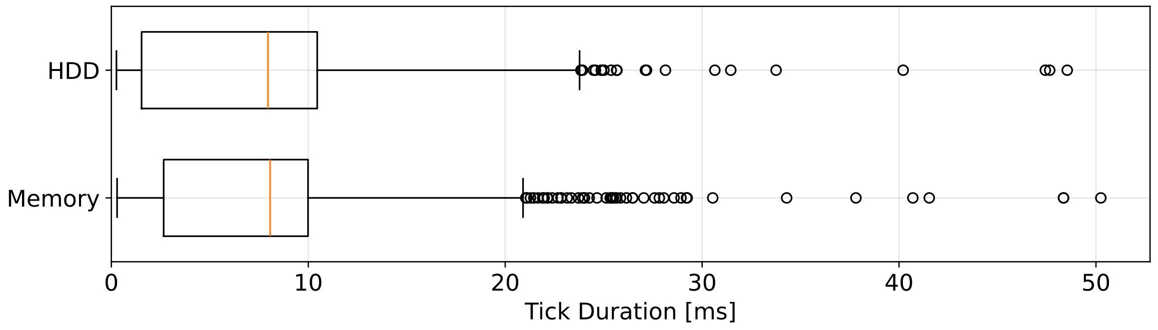
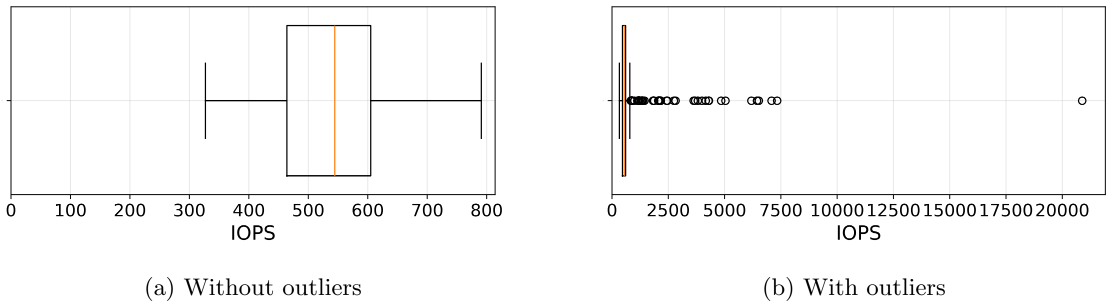


Figure 13: Walk around tick



(a) Without outliers

(b) With outliers

Figure 14: WalkAround IOPS

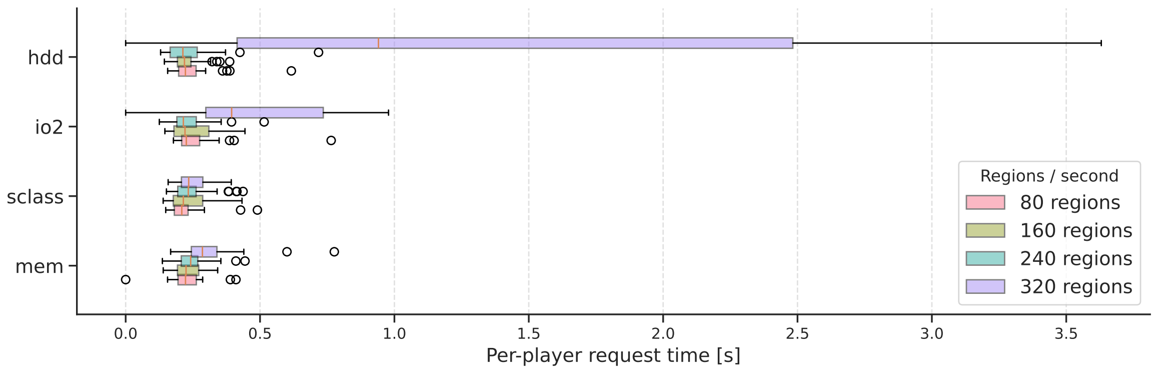


Figure 15: Join player latency per player

performance. We hypothesize that storage does not affect user-level performance due to the low storage request count compared to other workloads tested in this section. We show the IOPs both considering and not considering the outliers in Figure 14.

Observation 2. For stress-test read-trigger workloads as those are defined in Section 3.7, storage devices can improve experienced post-join lag; however, it does not influence player join times. To obtain the data, we spawned every player on the border of 4 regions. Therefore, whenever the player is spawned, they have to access region files corresponding to 4 unique regions, maximizing the number of unique files that need to be used. We, as well, put to the maximum the player render distance - 32 chunks - to maximize the number of file system accesses that need to be performed. Testing with an ideal storage case with memory-like throughput and latency, we find out that, in our setup, the maximum number of regions that can be loaded in a second is **320**. Attempting to run the workload with a higher region count leads to PaperMC crashing and disconnecting joining players. Testing for different storage devices, we observe a correlation between storage device and the file access latencies when a high number of regions are requested. We visualize

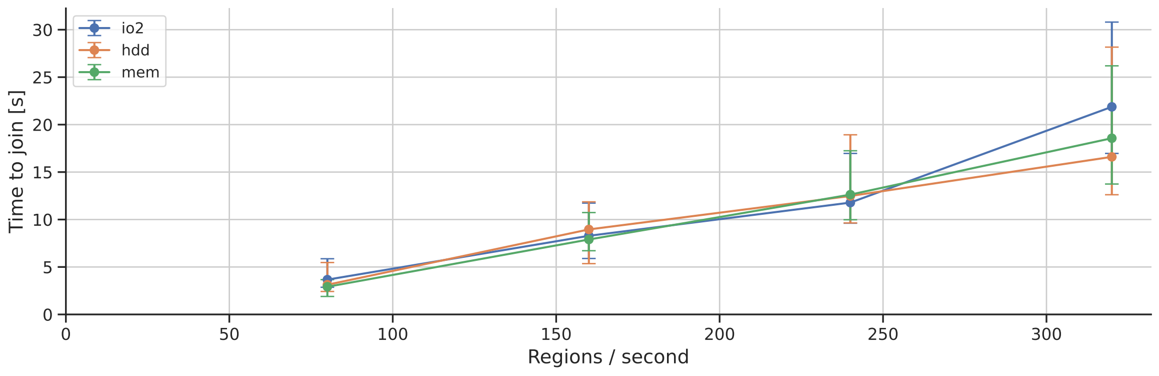


Figure 16: Median join times.

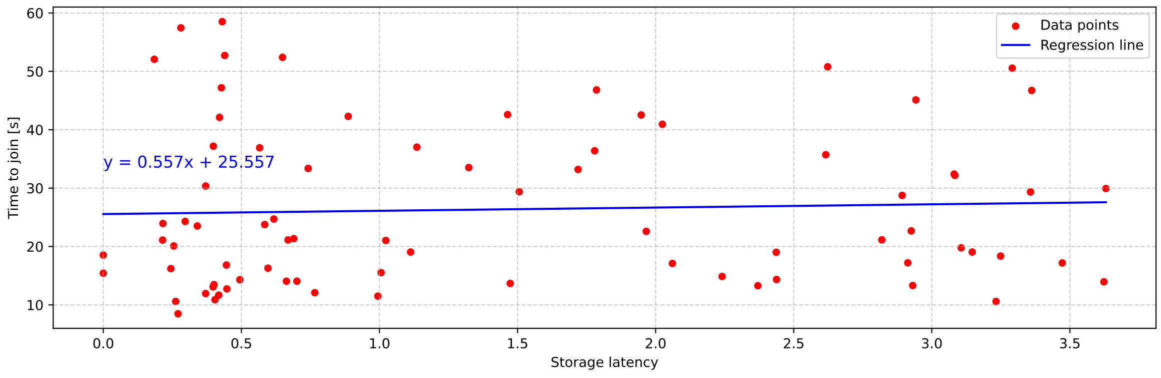


Figure 17: Dependency of join time from storage latency

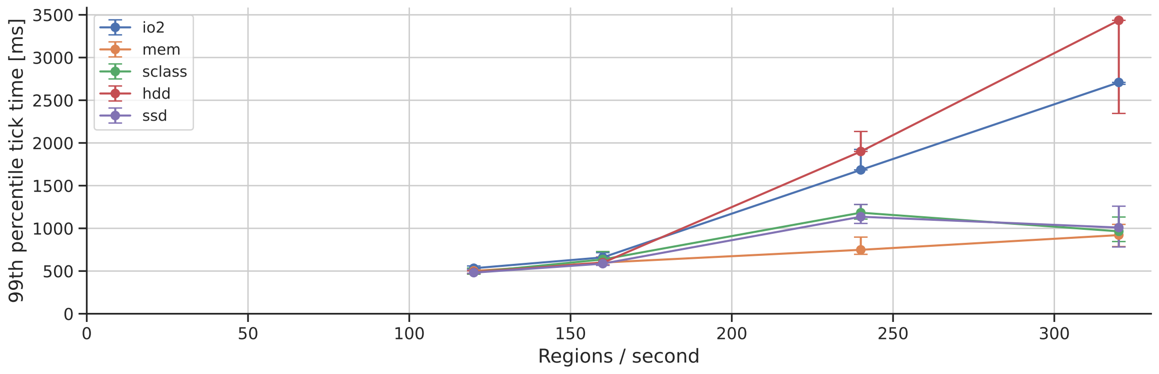


Figure 18: 99th percentile join server tick.

it in Figure 15. The difference peaks when requesting 320 regions/second, with HDD showing **3 times higher** median latency and IO2 showing **1.3 times** higher median latency than with baseline memory-like performance. However, when observing the join times, we can see that the join time increases for a higher number of regions accessed, but there is no difference between low- and high-performance storage devices. We show this in Figure 16 where we can also see that the experienced join time is significantly higher than storage request latency. There is, as well, a high variability in join times, making the storage device performance an insignificant factor on experienced player join time. To validate this finding, we asses the correlation between experienced player join times and the storage request latency for a particular region where the player is joining. We show it in Figure 17. While there is some correlation between experienced join time and storage request latency, the data appears

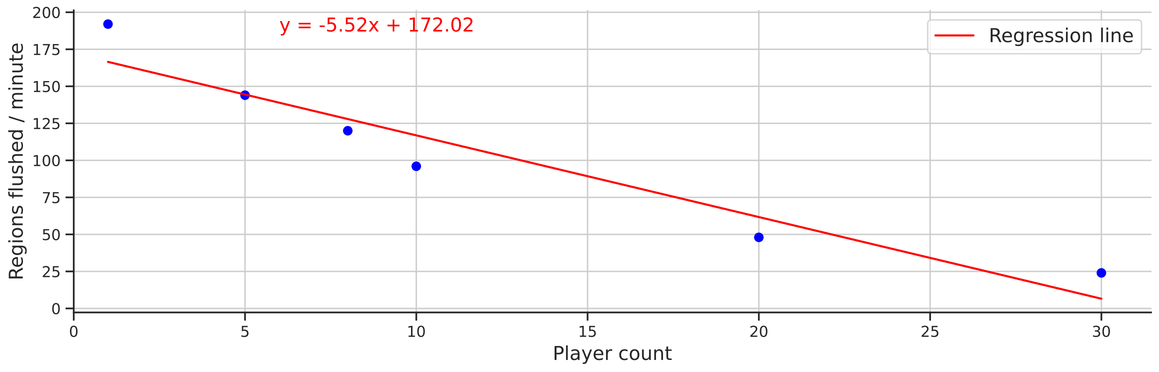


Figure 19: Correlation between participating players and the maximum teleportation per minute

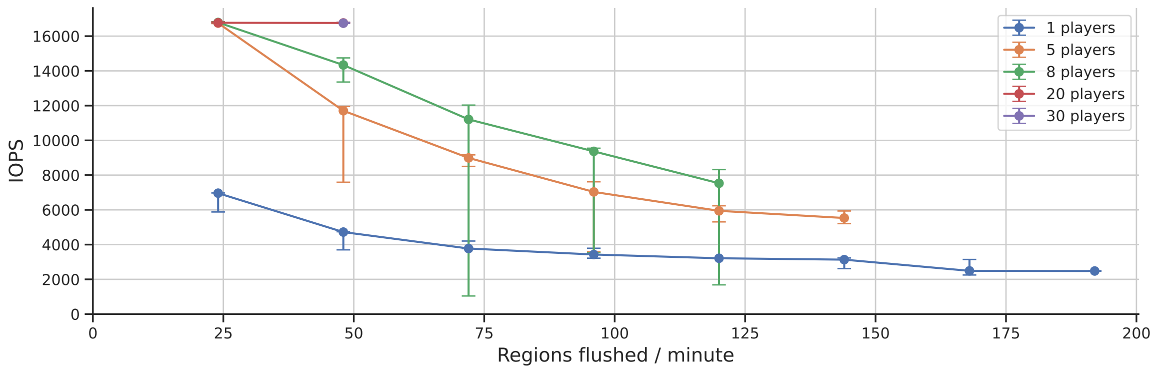


Figure 20: Per region IOPs from player count

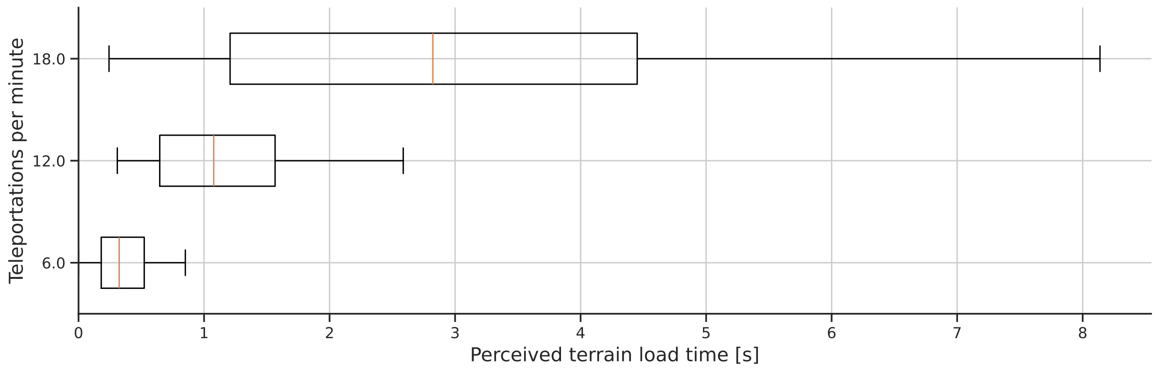


Figure 21: Teleportation region load time

to be mostly random. However, by measuring the tick time after join burst, we find out that the performance lag experienced after the burst can be decreased with high-performance storage devices. We show this in Figure 18. At high request region counts, the experienced performance lag expressed in 99th tick time percentile, can be decreased **3.5 times** from HDD and **2.7 times** from IO2 when comparing to the ideal memory-like storage case. There is **no difference** between SSD and faster storage devices and an ideal memory-like case.

Observation 3. In our setup, storage has no effect on MVE scalability for instantaneous write workloads as those are defined in Section 3.7. To the data, we first find the maximum number of storage writes that can be achieved under teleportation player behavior. In teleportation player behavior, players are teleported in a round-robin fashion at a pre-defined frequency. In our

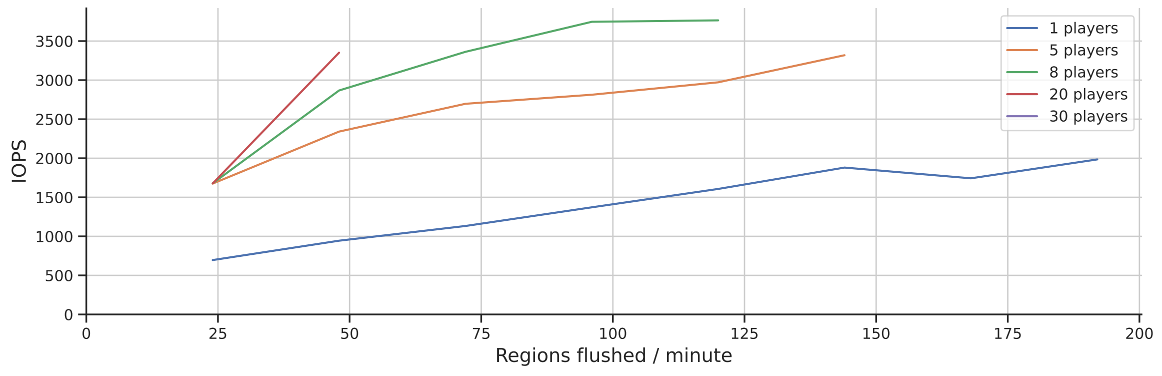


Figure 22: IOPs during workload

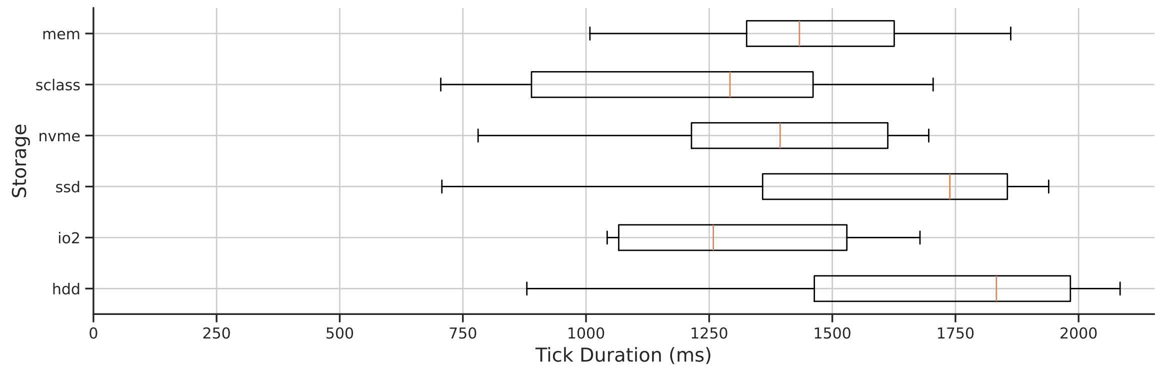


Figure 23: Storage vs tick time for teleportation

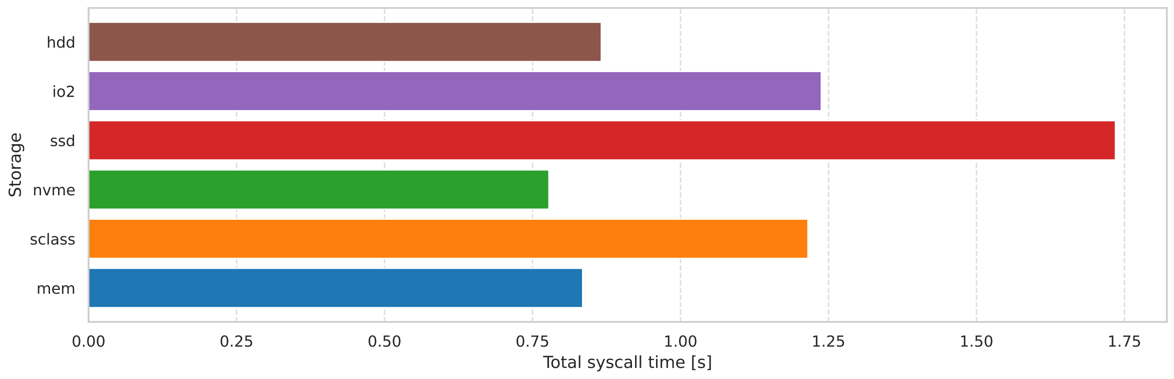


Figure 24: Total request latency

setup, to maximize the number of unique files in the workload, we teleport every player exactly on the border of 4 regions. To maximize the number of file system accesses, we set the render distance for each player to maximum. We find out that on ideal memory-like storage, as the number of players increases, the highest teleportation frequency at which the server can operate without crashing decreases. We show this in Figure 19. However, as the number of players decreases, so does the number of file system accesses executed when flushing each region. We show this in Figure 20. The reason is that players do not have enough time to generate the full region, which is also reflected in the experienced terrain load times that we summarize in Figure 21. Combining both the number of storage accesses executed per region and the maximum number of teleportations that the system can handle, we calculate the IOPs executed during the region flushes. We visualize it in Figure 22. There, we observe that the access peak for 8 players flushing 120 regions a minute. We take this data point, and for it, we record the system

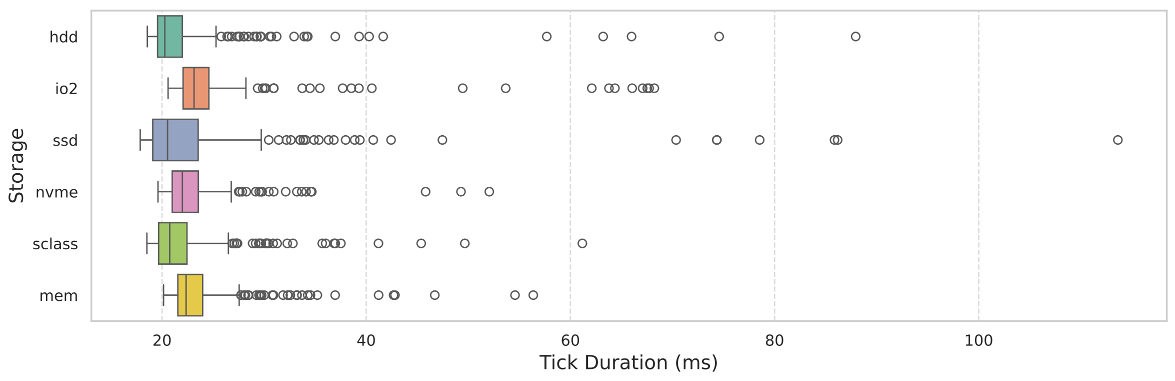
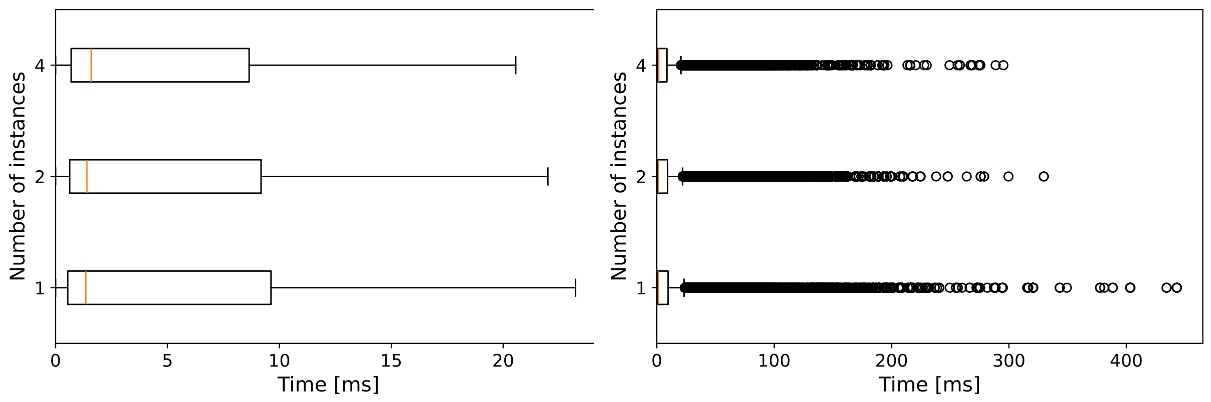


Figure 25: Tick duration during NPC synchronization



(a) Without outliers

(b) With outliers

Figure 26: Multi-instance tick time.

performance on different storage devices. We find that there is both no difference in system-level file access latency experienced when the system is flushing the regions and no difference in user-level metrics. We compare the total file access latency in Figure 24, and we compare server tick time in Figure 23.

Observation 4. Storage has no effect on MVE scalability for delayed write workloads. As discussed in Section 2.1, PaperMC limits the throughput and increases the buffer flushing interval for higher player counts. Therefore, to test MVE scalability under storage, we select the smallest number of player count to reduce the performance overhead not related to storage performance. We execute, on different storage devices, player NPC attack behavior with a total of 5 players. Maximizing the number of unique files accessed, we spawn every player on the border of 4 regions. We validate the results against different player counts and do not find any performance difference as well. To test the scalability, we observe the storage tick for 10 seconds after the synchronization point - the moment when PaperMC starts flushing NPC updates to the storage. Our results show that during the NPC synchronization, there is no difference in the server tick time Figure 25.

4.4 Multi Instance Performance

In this section, we try to collocate multiple PaperMC server instances on the same storage drive. Our motivation is the results of the Section 4.3 showing that there is little correlation between the storage system-level performance and the user-level QoS. The results are important because running multiple instances on the same storage drive decreases the number of drives required through resource sharing, which is popular in cloud-based deployments. To obtain the results, we run the copies of Yardstick-defined WalkAround [29] workload with **10** players spawning in the center of the maps and walking around in a pre-defined perimeter. In our experimentation we test collocating 1, 2, and 4 instances, with each instance getting 2 CPU cores and 4GB of RAM. For experimentation, we chose the storage configuration with the lowest QoS as described in Section 3.4.

Observation 5. Up to four MVE instances can be collocated on the same storage drive. We show the results obtained for a different number of collocated instances in Figure 26. There, we visualize the distribution of obtained server tick times. We observe that there is no correlation between the number of instances run and the user-level performance. We hypothesize that this is a result of high-levels of caching due to the players being located in overlapping regions as discussed in Section 2.3 and a low correlation between storage system-level performance and user-level performance for every PaperMC instance as discussed in Section 4.3.

5 Conclusion

In this paper, we investigated the correlation between the storage-level performance and MVE user-level QoS. We began our exploration by studying the correlation between the user actions and the storage accesses, and, based on collected data, we constructed a user-storage interaction model. Then, using the model, we constructed an MVE storage benchmark - PokeSto - and published it as an open-source artifact on GitHub. Finally, we used that benchmark to evaluate the effect of system-level performance on MVE user-level performance for a popular MVE server implementation - PaperMC.

5.1 Answering the research questions

RQ1 **What is the relation between user interactions and MVE storage accesses?** We found that MVE interacts with storage through a number of **I/O routines**. These routines are triggered as a result of the executed player actions. For example, a player moving into a new region will trigger associated read I/O routines. Each either reads or writes data to the storage. Writes can be either **instantaneous** or **delayed**. Delayed writes happen according to some **policy** and can adapt the delay and throughput based on server load. In case a similar to Minecraft **header-subregion** file structure is employed, MVE will show **random** accesses of **request size** dictated by the file structure.

RQ2 **How to design and implement a benchmark to evaluate MVE performance based on our storage model?** To design and implement a storage MVE benchmark, we continuously iterated over the *(i)* problem, *(ii)* benchmark requirements, and *(iii)* benchmark design. The result, published as a GitHub open-source artifact, is a generalizable benchmark MVE with trace-based workloads and storage simulation components.

RQ3 **What is MVE scalability on different storage devices?** Assessing the scalability of popular MVE server implementation - PaperMC - we find, in **observations 1-4**, that there is little correlation between the system-level performance and user-level performance, even for stress-test workloads. We, as well, in **observation 5**, find that at least up to **four** PaperMC servers can be collocated without any loss in performance.

5.2 Threats to validity

Limited scope of storage simulation. Real-world storage devices do not show a linear increase in response latency with an increase in the number of storage requests [13]. This is not captured by our storage simulation. As well, storage devices like NVMe or storage-class memories have several request and response queues in comparison to one simulated by **nullblk** [13]. Finally, storage throughput and latency can vary over time, which is not captured by our storage simulation.

Limited player behavior scope. We have picked, for convenience, only a small subset of player behavior to be tested. However, there can be other types of player behavior having a different effect on the storage, resulting in a different storage to user-level performance translation.

5.3 Future research

Improved storage simulation. The validity of the results would benefit from improved storage simulation that would capture the real-world storage throughput and latency performance variability and nonlinear scaling of response latency from the number of requests.

Large-scale player behavior archive and exploration The completeness of the results would benefit from exploration of a larger set of player behaviors. The project would benefit from real-world player behavior traces, which could be used to assess how the storage-level performance translates to user-level performance under real-world conditions.

References

[1] Chunk format – minecraft wiki, . URL https://minecraft.fandom.com/wiki/Chunk_format#.

[2] Region file format, . URL https://minecraft.fandom.com/wiki/Region_file_format.

[3] microsoft/DirectStorage, . URL https://github.com/microsoft/DirectStorage. original-date: 2022-02-01T17:59:52Z.

[4] Newzoo global games market report 2022 | free version, . URL https://newzoo.com/resources/trend-reports/newzoo-global-games-market-report-2022-free-version.

[5] SCM - storage class memory, . URL https://forum.huawei.com/enterprise/en/discuss-the-integration-of-virtualization-with-storage-solutions/thread/746561636157767680-667213859733254144.

[6] Tutorials/parkour – minecraft wiki, . URL https://minecraft.fandom.com/wiki/Tutorials/Parkour.

[7] PaperMC. URL https://papermc.io/.

[8] Amazon Web Services. Amazon efs performance specifications, 2025. URL https://docs.aws.amazon.com/efs/latest/ug/performance.html. Accessed: 2025-07-11.

[9] BittWare, a Molex Company. Ddr4 and ddr5 performance comparison, plus gddr6 and hbm2, 2024. URL https://www.bittware.com/resources/ddr4-and-ddr5-performance-comparison/. Accessed: 2025-07-11.

[10] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H C Du. Characterizing, modeling, and benchmarking RocksDB key-value workloads at facebook.

[11] PrismarineJS Contributors. mineflayer: Create minecraft bots with a powerful, stable, and high level javascript api. https://github.com/PrismarineJS/mineflayer, 2025. Version 4.30.0, MIT License.

[12] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM. ISBN 978-1-4503-0036-0. doi: 10.1145/1807128.1807152. URL https://dl.acm.org/doi/10.1145/1807128.1807152.

[13] Krijn Doekemeijer, Nick Tehrany, Balakrishnan Chandrasekaran, Matias Bjørling, and Animesh Trivedi. Performance characterization of nvme flash devices with zoned namespaces (zns). In *2023 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 118–131. IEEE, 2023.

[14] Dogcraft Community. Elytra races, 2025. URL https://dogcraft.net/wiki/Elytra_races. Accessed: 2025-07-11.

[15] Fujitsu Technology Solutions GmbH. Data sheet: Hard disk drive for fujitsu desktop pc esprimo and workstation celsius. Technical report, Fujitsu Technology Solutions GmbH, 2024. URL https://sp.ts.fujitsu.com/dmsp/Publications/public/ds-hard-disk-drives.pdf. Accessed: 2025-07-11.

[16] GigeNET Team. Nvme vs sata ssd speed: Which drive technology is faster in 2025?, 2025. URL https://www.gigenet.com/blog/nvme-vs-sata-ssd-speed-comparison/. Accessed: 2025-07-11.

[17] InfluxData. Building a telegraf plugin to collect data from your minecraft server, 2018. URL https://www.influxdata.com/blog/mineflux-monitoring-your-minecraft-server-with-influxdata/. Accessed: 2025-07-11.

[18] Alexandru Iosup, Fernando Kuipers, Ana Lucia Varbanescu, Paola Grosso, Animesh Trivedi, Jan Rellermeyer, Lin Wang, Alexandru Uta, and Francesco Regazzoni. Future computer systems and networking research in the netherlands: A manifesto. *arXiv preprint arXiv:2206.03259*, 2022.

[19] Linux kernel contributors. Null block device (null_blk), 2024. URL https://docs.kernel.org/admin-guide/blockdev/null_blk.html. Accessed: 2025-07-11.

[20] Chris Mellor. Intel confirms optane dimm and ssd speed, 2018. URL https://blocksandfiles.com/2018/12/13/intel-confirms-optane-dimm-and-ssd-speed/. Accessed: 2025-07-11.

[21] Vlad Nae, Alexandru Iosup, and Radu Prodan. Dynamic resource provisioning in massively multiplayer online games. 22(3):380–395. ISSN 1045-9219. doi: 10.1109/TPDS.2010.82. URL http://ieeexplore.ieee.org/document/5444882/.

[22] Mihir Nanavati, Malte Schwarzkopf, Jake Wires, and Andrew Warfield. Non-volatile storage: Implications of the datacenter's shifting center. 13(9):33–56. ISSN 1542-7730, 1542-7749. doi: 10.1145/2857274.2874238. URL https://dl.acm.org/doi/10.1145/2857274.2874238.

[23] Felix Richter. Infographic: Are you not entertained? URL https://www.statista.com/chart/22392/global-revenue-of-selected-entertainment-industry-sectors.

[24] Seagate Technology LLC. Barracuda 2.5-inch hdd data sheet. Technical report, Seagate Technology LLC, 2019. URL https://www.seagate.com/www-content/datasheets/pdfs/barracuda-2-5-DS1907-2-1907US-en_US.pdf. Accessed: 2025-07-11.

[25] M. Seltzer, D. Krinsky, K. Smith, and Xiaolan Zhang. The case for application-specific benchmarking. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, pages 102–107. IEEE Comput. Soc. ISBN 978-0-7695-0237-3. doi: 10.1109/HOTOS.1999.798385. URL http://ieeexplore.ieee.org/document/798385/.

[26] ServerSimply Team. Comparing sas, sata, nvme, and cxl ssds in 2025: A comprehensive guide for sysadmins, 2025. URL https://www.serversimply.com/blog/comparing-sas-sata-nvme-and-cxl. Accessed: 2025-07-11.

[27] Statista. Metaverse: market data & analysis. https://www.statista.com/study/132822/metaverse-market-report/, 2024. Projected metaverse market size to reach over $500 billion by 2030.

[28] strace Contributors. strace: A diagnostic, debugging and instructional userspace utility for linux. https://strace.io, 2025. Version 6.7, GNU LGPL v2.1 or later.

[29] Jerom Van Der Sar, Jesse Donkervliet, and Alexandru Iosup. Yardstick: A benchmark for minecraft-like services. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, pages 243–253. ACM. ISBN 978-1-4503-6239-9. doi: 10.1145/3297663.3310307. URL https://dl.acm.org/doi/10.1145/3297663.3310307.

[30] vortexnl. Tick lag (1000 - 1500ms) every 45 seconds due to 'world save', is this normal? URL www.reddit.com/r/feedthebeast/comments/u6b2bw/tick_lag_1000_1500ms_every_45_seconds_due_to/.

[31] Baek Youngkyun. Mining educational implications of minecraft. URL https://www.tandfonline.com/doi/epdf/10.1080/07380569.2020.1719802?needAccess=true. ISSN: 0738-0569.

6 Appendix

6.1 Abstract

GitHub repository

6.2 Artifact check-list (meta-information)

Obligatory. Use just a few informal keywords in all fields applicable to your artifacts and remove the rest. This information is needed to find appropriate reviewers and gradually unify artifact meta information in Digital Libraries.

- **Program: Yardstick, Nullblk**
- **Compilation: none. Python 3.9 used of interpretation**
- **Binary: none**
- **Run-time environment: QEMU**
- **Hardware: Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz, 256 GB**
- **Execution: Python**
- **Metrics: storage, MVE Quality of Service**
- **Experiments: MVE single instance scalability, MVE multi instance scalability**
- **How much disk space required (approximately)?: 5 GB**
- **How much time is needed to prepare workflow (approximately)?: 0**
- **How much time is needed to complete experiments (approximately)?: user-defined. At least 5 minutes**
- **Publicly available?: Yes**
- **Code licenses (if publicly available)?: None**
- **Data licenses (if publicly available)?: None**
- **Workflow framework used?: Ansible**
- **Archived (provide DOI)?: No**

6.3 Description

How to access

GitHub

Software dependencies

Python, Jupyter, Nullblk