Vrije Universiteit Amsterdam

Honours Programme, Research Thesis

Does Your Server Need SSD? Applying General Purpose Minecraft-Like Games Benchmark Framework to Study Storage Performance Implications

---

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

1st supervisor:	Jesse Donkervliet	(VU Amsterdam)
2nd supervisor:	Krijn Doekemeijer	(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

November 19, 2024

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 [26]. Modifiable virtual enviroments (MVEs) games is an emerging game sub-genre with persistence functional requirements. Most popular MVE game - Minecraft - is played by over 140 million people monthly [9], and the Metaverse - a prominent modifiable virtual environment application - market is expected to grow up to 507 billion dollars by 2030. While storage can and does affect gaming performance and is central for MVE's persistent information storage, there yet is no investigation in the way it is used by MVEs. There yet is no available MVE storage traces, no information on how storage can impact MVEs user's Quality of Service (QoS), and there is yet no tool available to measure the impact. To measure a new performance aspect, it, as well, requires in-depth modification of existing performance measurement tools. Current work outlines a need for application-specific storage effect performance investigation [27] for application-specific benchmark for storage performance effect investigation [27] on a comprehensible set of application workloads. Current benchmarks such as [29] and [22] do not focus on recording storage performance and provide limited statically-defined workloads. This paper covers this gap. We start by defining TheStick - a framework for fully customized MLG benchmark. Then we model user MLG storage interaction. After, we apply TheStick framework to study storage effect onto user Quality of Service (QoS). Finally, from the results we derive a set of actionable conclusions.

Contents

1 Introduction **5**

2 Research Questions **6**

3 Background **7**
3.1 Storage Model . . . 7
3.2 Minecraft-like Games Behavior Model . . . 8
3.3 Minecraft-Specific Data Storage Implementation Details . . . 8
3.3.1 Files Hierarchy . . . 8
3.3.2 Region File Format and Terrain Structure . . . 9
3.3.3 World Saving Mechanics . . . 9
3.3.4 Point of Interest Data . . . 9
3.4 Minecraft Player Preferential Attachment . . . 10

4 Design of TheStick - A General Purpose MLG Benchmark Framework **11**
4.1 Framework Use Cases . . . 11
4.2 Framework Requirements . . . 11
4.3 Defining Actions - A Tool to Represent Arbitrary Complex User Behavior . . . 12
4.4 Framework Design . . . 12
4.4.1 Design Overview . . . 12
4.4.2 Player Emulation Design . . . 14
4.4.3 Metrics Collector Design . . . 15
4.5 Customizing Storage-Stick for Your MLG Server . . . 15
4.5.1 Describing Specific to Your MLG Behavior: Adding Your Own Action . . . 15
4.5.2 From an Action to a Customized Workload: Composing Own Workload . . . 15
4.5.3 Recording Relevant to You Metrics: Adding Your Own Metric . . . 15
4.5.4 Adding Your Own MLG . . . 15

5 Modeling MLG User Storage Interaction **16**
5.1 Defining What Data Minecraft-Like Games Store: Data Model . . . 16
5.1.1 Terrain Data . . . 16
5.1.2 Non-Playable Character Data . . . 17
5.1.3 Player Data . . . 17
5.2 Defining How Minecraft-like Games Store: Storage Process Model . . . 17
5.3 Actions - Instrument for Modeling User Interactions . . . 18
5.3.1 The Definition and Goals of MLG Actions . . . 18
5.3.2 Action List Requirements to Target Workload Composability . . . 18
5.3.3 Action Requirements for Modeling . . . 19
5.4 From User Interaction to Storage: Modeling Storage Interaction Through Actions . . . 19
5.4.1 Constructing a Model . . . 19

6 Applying TheStick Framework to Study Storage Impact on Quality of Service - StorageStick Benchmark **21**
6.1 Requirements . . . 21
6.2 Metrics . . . 21
6.2.1 System-Level Metrics . . . 21
6.2.2 Application-Level Metrics . . . 22
6.2.3 Collection Methodology . . . 23
6.3 Workloads . . . 23
6.3.1 Players Joining the World . . . 23
6.3.2 Fighting: Attacking NPCs . . . 23
6.3.3 Fighting: Attacking Other Players . . . 23
6.3.4 Modifying Terrain: Building Blocks . . . 23
6.3.5 Modifying Terrain: Removing Blocks . . . 23
6.3.6 Modifying Terrain: Changing Redstone . . . 23

6.3.7 Exploring The World . . . 23
6.3.8 Workload 2 . . . 23
6.3.9 Workload 3 . . . 23
6.4 Systems . . . 23

7 Evaluation **24**
7.1 Experiment Storage Systems . . . 24
7.2 Main Findings . . . 24
7.3 Players Joining and Leaving the World . . . 24
7.4 Findings for Players Joining the World . . . 24
7.5 Players Exploring . . . 24
7.6 Players Fighting . . . 24
7.7 Finding 2 . . . 24
7.8 Finding 3 . . . 24

1 Introduction

The gaming industry is the world's largest entertainment industry - worldwide, games engage over 3.2 billion players [9] and yield over $192 billion in revenue [26]. 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 player's and world's states by attacking players, removing blocks, building blocks, interacting with NPCs, etc, and the changes are persistently stored. MVEs' biggest representative Minecraft played by over 140 million people around the globe [9] and the promiment class of MVEs application - Metaverse - is predicted to grow to 507 billion dollar market by 2030.

MVEs are particularly interesting to study due to their large economical impact shown by their current and expected market sizes, and social impact shown by their current use in recreation, education where it already used in primary school education for teaching science and foreign languages [30], and for employee training by companies such as KLM, KFC, and UPS. Moreover, by 2030 the Metaverse is expected to engage over 2.6 billion people through the possibilities of enhanced remote learning, working, social interaction, and more. MVEs have to support a fast-paced game with constant modifications which, due to persistence requirement, have to be stored on a persistent storage device. Shown by DirectStorage [8] - Microsoft API to optimize load time by increasing game assets decoding speed storage is a point of interest for gaming performance engineering community and can have impact on QoS.

However, despite storage being essential for MVE functionality and storage having a potential impact on application QoS, there is still no investigation on storage MVE use and impact on MVE performance or performance variability.

One of the storage performance engineering challenges is a nonlinear transition of performance improvements from a singular component to an end-to-end system. For example, a 10000 times latency improvement from HDD to storage class memories [10] results in only 7 times performance improvement for MySQL workloads [24]. The nonlinear performance improvement translation can be explained by the storage stack consisting of multiple software layers, such as file system, I/O scheduler, and others. Each of the software layers has own performance. Storage also interacts with multiple hardware components such as CPU for scheduling or memory for file system caching making final end-to-end performance translation unpredictable. Furthermore, whether a particular storage system change will have an impact on performance of an end-to-end system depends on the application logic itself and factors such as use of blocking or non-blocking file system calls and use of background threads for storage I/O operations. As a result, there has been a case made for application-specific storage benchmarking [27] and published storage application-specific benchmarks such as YCSB [21] or RockDB [20].

There has been work done on measuring Minecraft-like Games (MLGs) scalability and outlining issues with low performance with a large number of players [29]. There has also been a work on measuring impact of MLG workloads on instability ratio and outlining possible issues with regards to QoS guarantees. However, despite storage I/O operations being essential to MVE persistent properties, there has yet been no work done investigating MVE storage use and measuring storage impact on MVE performance or performance variability.

In this paper we cover this gap. We start by modeling MVE storage interaction, and build a benchmark - StorageStick - a storage benchmark for MVE. We analyze storage performance impact on Minecraft and, by collecting storage traces during benchark execution, analyze Minecraft storage use. Our paper shows that storage can have an impact on MVE QoS. We finish our paper with actionable insights for game developers and game operators for MVEs.

2 Research Questions

We begin answering a problem by defining research questions:

- **RQ1**. What are MLG storage access patterns triggered via user actions?
- **RQ2**. How to design and implement a benchmark to evaluate MLG performance based on used storage model?
- **RQ3**. How does MLGs user-level performance scale in terms of storage performance?
- **RQ4**. How do storage stacks compare in terms of MLG performance?

3 Background

In this section, we show Minecraft-Like Games and storage model used later in the paper, introduce and important for storage workloads property of player's preferential attachment and provide, necessary for further reading, background information on Minecraft-specific features including its world saving mechanics and file format.

3.1 Storage Model

In this section we introduce a storage model used throughput the paper for evaluating the impact of storage on Minecraft user-level performance. The goal for our model is to be as simple as possible and yet reflect important performance metrics and bottlenecks that can impact MLG quality of service. Reflecting on storage implications of player preferential attachment introduced in Section 3.4, we model storage as a two component system consisting of file system with read cache and write buffers, and storage medium with limited throughput, latency, and operation queues. We model these components as two separate as reads and writes from those incur orders of magnitude different latencies and throughput. As well, under different player clustering, two components will compose performance profile to a different extent - fast storage medium bears greater importance for servers with low player preferential attachment.

File system cache and buffers have a limited size. We model those in terms of their size because increasing cache or buffer size is a tradeoff between the memory consumption and a potential performance benefit. The correlation between amount of memory allocated for cache and buffer and the performance benefit depends on system workload and little memory addition can bring both high performance benefit or none. We model storage medium in terms of its latency, throughput and operational queues. We include latency and throughput as those are order of magnitude different to one of memory. All latency, throughput, and number and depth of operational queues can be within an order of magnitude difference [28] amongst comparable devices. Thus, we are interested to quantify the impact of those metrics on MLG quality of service.

MLG, when reading from storage will first attempt to read from file system cache unless specified otherwise via its open flags. If flags are indicating that the read should not be performed from cache, we call it a direct read. Indirect read will first attempt to read from cache, and only if the file is not present in cache, read will be performed from a storage medium. Direct reads will bypass cache reading directly from storage medium. Indirect writes will first attempt to write to buffer and only if the buffer is full, write would have to be performed directly to the storage. In case of direct write, one will be performed onto storage medium bypassing write buffer. We separate direct and indirect read and write operations as those, on average, will yield lower latency and provide higher bandwidth although their worst case matched direct reads and writes.

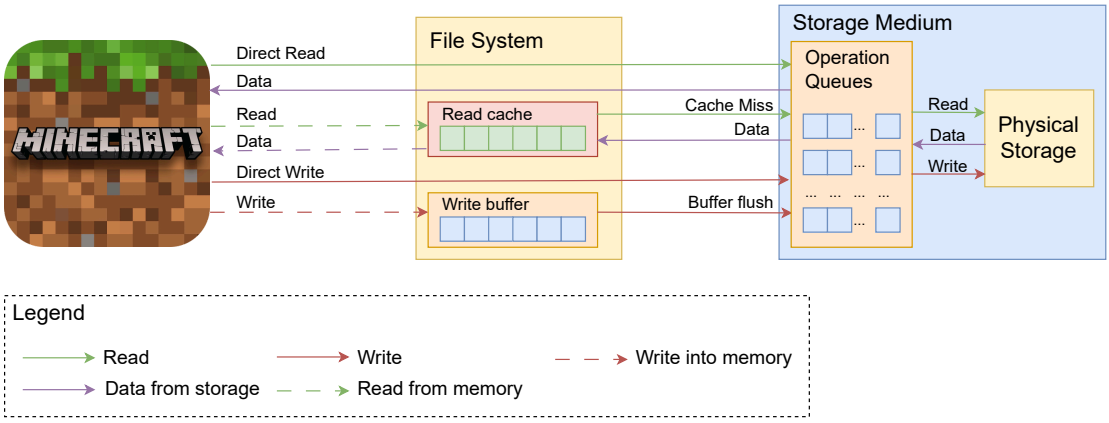


Figure 1: Storage Model

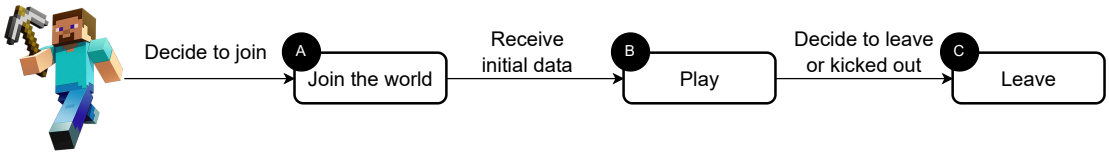


Figure 2: Session Lifetime

3.2 Minecraft-like Games Behavior Model

This section defined MLG user behavior model. To construct the model, similarly MLG data model (**??**), we use Minecraft, Terraria, and Roblox as references. To represent player behavior throughout their whole session, we subdivide this section into two parts - analyzing what kind of user behavior required "around" the MLG play to accommodate for it, and analyzing which kind of user behavior constitutes an MLG play.

We define the whole MLG player session to consist of three steps as summarized in fig. 2: Ⓐ player joining the world, Ⓑ playing, and Ⓒ player leaving the server. Ⓐ can solely be triggered via user's decision to join an MLG world. If joining is successful, the transition to Ⓑ happens automatically upon user receiving some initial state after which they can render and start interacting with the world. Finally, transition to Ⓒ occurs when either a player, themselves, decide to leave, or when they are kicked out of the game either due to server administration decision or due to internal game mechanics - for example as a result of user's poor internet connection. This three steps are evidently minimal to support any multiplayer MLG session: without Ⓐ it would be impossible to start MLG session, without Ⓑ there will be no session, and without Ⓒ there will no way to end the session.

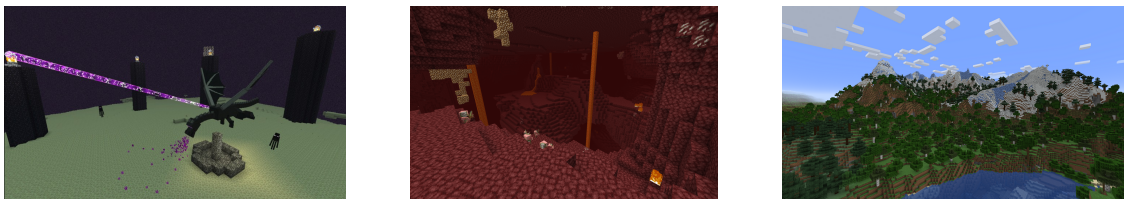
When defining MLG player behavior that constitutes MLG play, to ensure coverage and relevance of classified behavior, we refer to classification introduced by Müller et al. in Heapcraft **??** - mining, building, fighting, and exploring. Distinction between building and mining, where mining refers to a process of getting underground resources, is Minecraft-specific arising from Minecraft-specific game mechanics of gaining resources in survival mode. To make both building and mining applicable to a larger set of MLGs, we collapse those two into a terrain modification that is in some form present in all Minecraft, Terraria, and Roblox. Additionally, behavior allowing for terrain modification is dictated via modifiability property of MLGs. Exploring refers to simply a position change which is present in all Terraria, Roblox, and Minecraft and is a requirement of all "walkable" virtual environments. Finally, while possibility of fighting does not arise from any MLG requirements, we optionally include it in our behavior model as it is a popular game mechanic available in all Minecraft, Roblox, and Terraria. Fighting can be done either against NPCs or other players.

3.3 Minecraft-Specific Data Storage Implementation Details

This section applied only to Minecraft and describes necessary for further understanding of evaluation for both actions storage use characterization Section 5 and benchmark evaluation.

3.3.1 Files Hierarchy

Persistent Minecraft world data is located under *./world* directory. The further subfolders are organized into folder saving terrain data *./world/region*, non-playable-character (NPC) data in *./world/entities*, point of interest (POI) data in *./world/poi*, and player data in *./world/playerdata*. There are as well folders responsible for other dimensions: *./world/DIM-1* for the Nether, and *./world/DIM-0* both having the same substructure of NPC, region and POI subdirectories. All terrain, NPC, and POI data is organized into files based on the region (region is further defined in **??**) and follow a naming structure of *r.x.y.mca* where x is region coordinate along x-axis and y is region coordinate along y axis. Player data files stored under *./world/playerdata* follow the naming structure of *uuid.dat* where uuid is a unique player identifier derived from player's nickname.



(a) The End (b) Nether (c) Overworld

Figure 3: Minecraft Dimensions

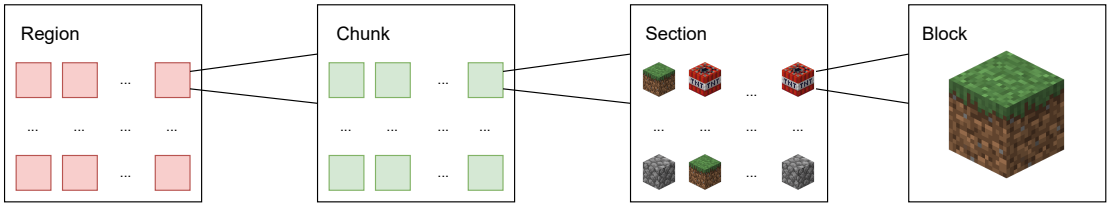


Figure 4: Region File Structure

3.3.2 Region File Format and Terrain Structure

Minecraft terrain and NPC data is divided into 3 dimensions - overworld, nether, and the end. Overworld (fig. 5) is a primary dimension where players initially spawn and spend the majority of the game. The nether is a dimension which players can get into via a nether portal and is used to get resources not available in the overworld. When player teleports to the nether, the coordinates of their spawn are proportional to their overworld position. Finally end dimension is one players travel to kill the final boss of the game - ender dragon. Players can reach it via the end portal and their spawn will always be at a fixed position [7].

Dimensions are divided into regions with every region represented via *.mca* region file. Every region is divided into 32 by 32 chunks [4]. Each chunk, in turn is an area of 16x256x16 blocks. To store information about each chunk, Minecraft files are divided into header and data parts where header contains information on chunk size and offset within the file and data contains per-chunk information on either blocks or NPC present and their properties. For reader's understanding, we summarize terrain hierarchy in fig. 4.

Minecraft terrain's minimal world unit is a block - all the terrain structures in Minecraft are made out of blocks. The next unit is a section which is an area of 16x16x16 blocks

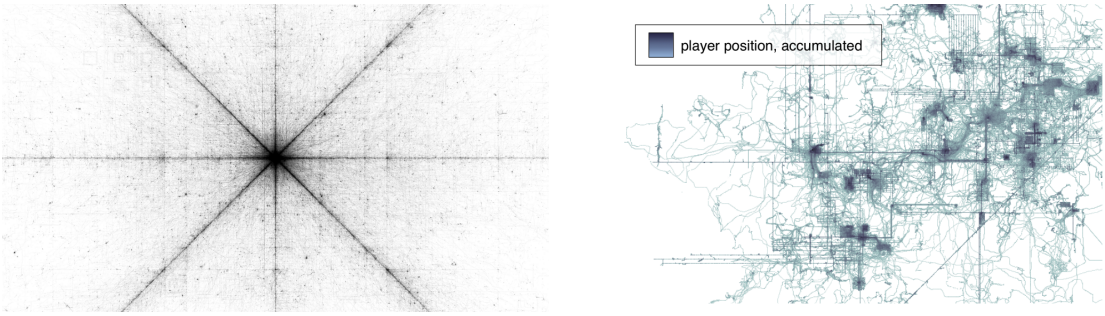
Minecraft terrain is stored under *./world/region* directory and is divided into 3 dimensions - overworld, nether, and the end. Overworld (fig. 5) is a primary dimension where players initially spawn and spend the majority of the game. Overworld region files are stored under *./world/region* directory. The nether is a dimension which players can get into via a nether portal and is used to get resources not available in the overworld. Nether region files are stored under *./world/DIM-1/region* folder. When player teleports to the nether, the coordinates of their spawn are proportional to their overworld position. Finally end dimension is one players travel to kill the final boss of the game - ender dragon. Players can reach it via the end portal and their spawn will always be at [0, 0] region [7].

Every Minecraft region is represented by one *.mca* file and every region is subdivided into chunks: every region is 32 by 32 chunks [4]. Every chunk is up to 16 sections [6]: the sections are vertically stacked on top of each other. Ever section is area of 16x16x16 blocks. Finally, every block has its own state. The hierarchy is summarized in fig. 4.

3.3.3 World Saving Mechanics

3.3.4 Point of Interest Data

For some structures like beehives, beenests, nether portal, etc, Minecraft saves those into the point of interest files [3]. Every point of interest files contains a set of records and every record is a position



(a) 2b2t Global Distribution

(b) Local Distribution

Figure 5: Player distribution

and a type of point of interest.

3.4 Minecraft Player Preferential Attachment

In this section, we introduce a notion of Minecraft player's preferential attachment - a property describing how close players play to each other in Minecraft-like Games world. Different servers can have different degrees of players preferential attachment and the resulting tool - StorageStick - and findings are applicable to all. However, different degrees of player's preferential attachment would result in different storage bottlenecks for storage system modeled in Section 3.1.

Preferential attachment is a graph theory property applicable to a large number of real-world networks. It states that, as networks evolve, when a new node is added, it is more likely to attach itself to another node that already has a large number of connections [25]. We adapt this concept for MLGs defining it as: when a player joins a Minecraft world, they are more likely to join areas with other players already playing. For MLGs, we differentiate between local and global preferential attachments. Global preferential attachment is summarized on 2b2t heatmap (fig. 5a). 2b2t server - a server over 780,000 unique players over the course of its history [19] - in the period from 2018 till 2021 during nocom hack [1]. The heatmap visualized player's activity for blocks from -245K to 245K with darker higher signifying the higher levels of activity. We explain higher levels of activity in the center of the map as a result of players spawning in the center of the map and taking progressively longer time to reach further map points. Local preferential attachment is visualized by Muller et al. fig. 5b where every pixel is one Minecraft block. We explain it via the existence of points of interest such as villages, mining shafts or others around which players concentrate their activity.

This is an important property for our work because, as summarized in Section 3.1, storage requests can be served both from storage or cache serving storage requests from memory. If storage request is served from cache, then it will incur lower latency. If a server has high degree of player's preferential attachment, the more requests will be served from cache, and the less storage medium will be a performance bottleneck.

4 Design of TheStick - A General Purpose MLG Benchmark Framework

In this section we present a design of TheStick - a general purpose framework for MLG benchmarking. We designed this framework after an idea of every user being able to get results as relevant as possible to their particular use case. This framework is later applied to StorageStick - a benchmark made on top of TheStick architecture to measure impact of storage on QoS in Section 6. To achieve that, we started by outlining two use cases for both *i) server operators with already running server* and *ii) server operators who yet do not have a running server* in Section 4.1. Based on use-cases, we derive a set of framework requirements in Section 4.2 and based on requirements we present a design in Section 4.4. The main novelty of our framework is full customization of executed workload through *actions* - an instrument we define in Section 4.3. As part of design we, as well, provide instructions on how server operator can customize *j) their workload*, *jj) MLG benchmark is executed on*, and *jjj) metrics recorded* to best suit their application of TheStick framework.

4.1 Framework Use Cases

To comprise a list of requirements, we, first summarize the use cases under which we would like to see *TheStick benchmark* used. *TheStick benchmark* is any benchmark implemented on top of TheStick framework design summarized in Section 4.4. With framework design, we target primarily two user groups: *i) server operators that already have their MLG server* and *ii) server operators who do not yet have their server*. We want to design framework in such a way that *i)* can benefit from data collected from their server in benchmarking their particular use-case benefiting from most-relevant to them results. Meanwhile, *ii)* should not be restricted from running the benchmark via lack of data and user still should be able to get relevant to them performance results. From these two groups, we define two primary benchmark use cases summarized in fig. 6. Case Ⓐ is a use case applicable to *i)* where user first gets *user activity trace* and *persistent world data* and then uses it as input to *TheStick benchmark* getting the results. Case Ⓑ, on the other hand, is applicable to *ii)* where user directly calls *TheStick benchmark* controlling it through some list of paramters and gets the results.

4.2 Framework Requirements

With our benchmark design we are trying to provide support to both users who *i) have data and resource for benchmark customization for their needs* and *ii) ones who do not have data or resources to customize benchmark for their needs*. An example representative from *i)* can be a large scale server operator having a record of user activity of a particular server and wanting to test whether a particular

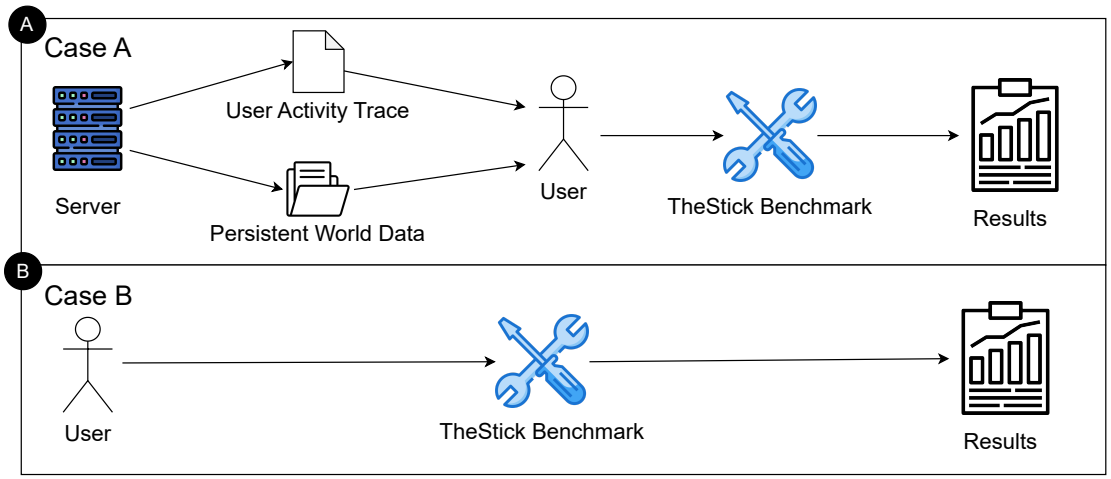


Figure 6: Framework Use Cases

hardware upgrade would bring them a worthwhile performance benefit. An example representative of *ii)* can be a prospective server operator who does not yet have record of user activity and wanting to figure out the initial hardware configuration based on anticipated player count and workload. While serving the interest of both groups we still want framework to be applicable to any MLG and be provide reproducible results. To satisfy all, we pose a list of framework requirements (FR):

FR1 *Full Customization*: framework should allow customization for a specific to server operator use-case without any significant modifications to framework architecture.

FR2 *No Mandatory Prerequisites*: framework should provide a sufficient set of helper utilities to avoid mandating user to have any data at hand for benchmark execution.

FR3 *MLG Generalization*: framework should be applicable to all MLGs and in its implementation avoid favoring any system.

FR4 *Reproducibility*: the resulting modifications to the virtual world should be persistent on every run.

4.3 Defining Actions - A Tool to Represent Arbitrary Complex User Behavior

To follow a requirement **FR1** of *customization* and with a goal of allowing server operators to represent an arbitrary complex user behavior replicating ones on their servers, we introduce a notion of *action*. An *action* is an arbitrary user interaction that follows two framework action requirements (FAR):

FAR1 *Sequential*: the actions of an individual player can only be executed in sequential manner at most one at a time.

FAR2 *Deterministic*: an action can be executed only in one way and lead to the same virtual world modifications.

FAR3 *Traceable*: server operator should, in some way, be able to record an action on their server.

Our idea behind actions is that, since real-world players, similar to our modeling, can execute only one actions at a time, a sum of sequential actions can represent an arbitrary complex real-world workload. With actions, due to their traceability property, a server operator can replay a recorded user activity getting the most relevant metrics for their particular use case. Due to deterministic property the result of every replay will be consistent.

4.4 Framework Design

In this section, we present a design overview of framework. In particular, we focus on high level picture to provide a reader with intuition of framework inner-workings in Section 4.4.1 and we focus on player emulation component responsible for implementing a core framework concept - actions from Section 4.3 - in Section 4.4.2.

4.4.1 Design Overview

To fulfill requirements posed in Section 4.2, we propose framework design summarized in fig. 7. This framework supports two modes of operation: *i) customized* that allows server operators to supply their own user activity and virtual environment data and *ii) synthetic* that generates synthetic traces and virtual world data to run benchmark on. The user can also use a combination of two operation modes and for example supply their own trace of user activity but use one of utilities provided to generate synthetic world data. Presence of both tools for customization and not mandating their use by introducing utilities makes framework satisfy requirements **FR1** and **FR2**.

To support a customization of workloads, the design revolves around the idea of Ⓒ trace file that is a set of action records. Each action contains a timestamp for its execution, action name self, and arguments the action shall be executed with. We give additional instructions, including trace file format, on customization of trace file in a larger workload in Section 4.5.2. The trace file is

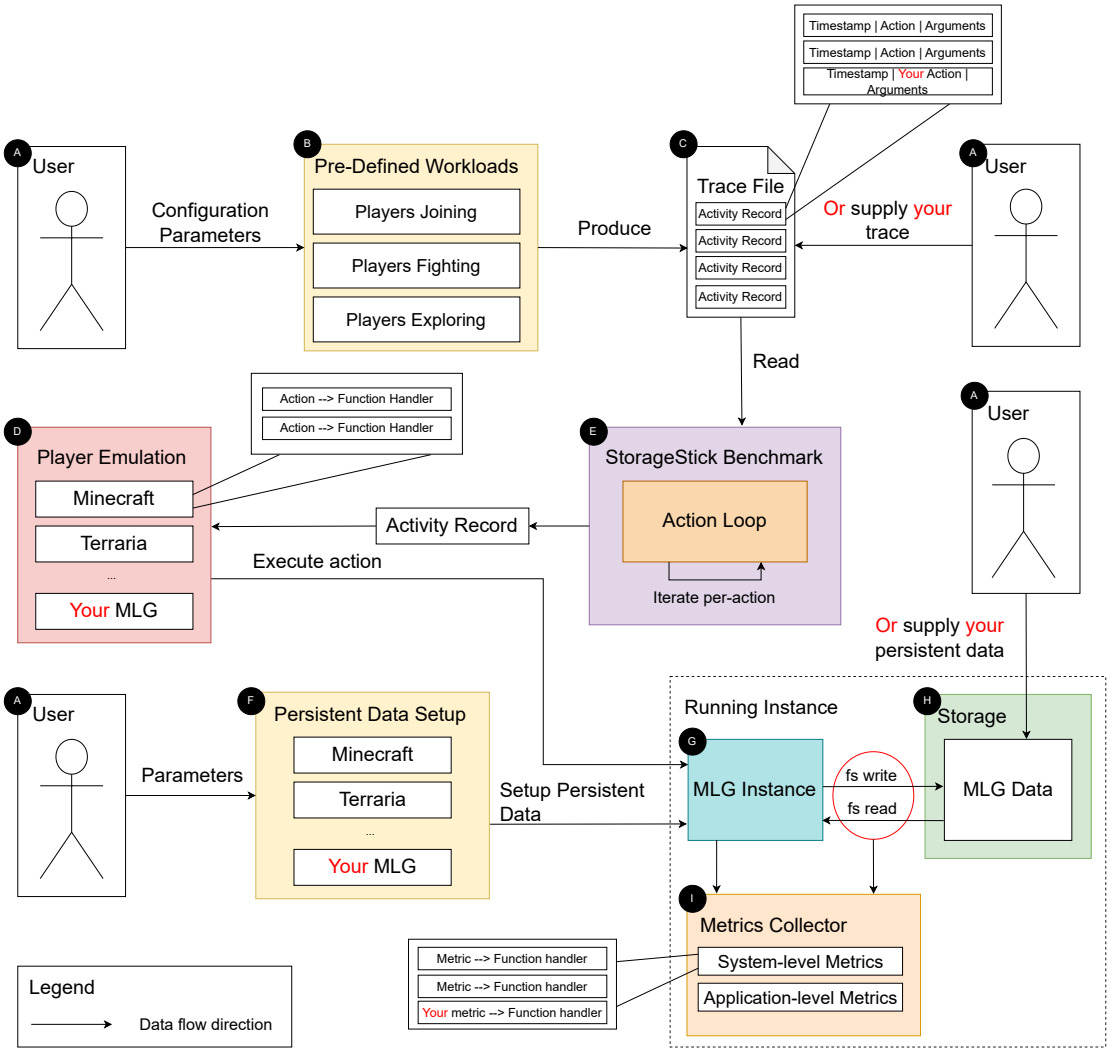


Figure 7: Storage Stick Design Overview

read line by line by (E) benchmark executor that is busy-looping for time to match the action record timestamp when action is passed to the (D) player emulation component. Player emulation contains, for each MLG, a mapping from each action to a function handler that executes the action on (G) MLG instance using appropriate to MLG protocol. As well, it ensures the atomicity of each action executed. The inner-workings of player emulation component are further described in Section 4.4.2 and the instructions towards adding your own actions are presented in Section 4.5.1. To support customization of modifiable virtual environment, we allow users to upload their virtual environment data for use. The data is copied and placed onto (H) storage for execution at the beginning of each benchmark execution ensuring *reproducibility* (**FR3**). To support customization of collected metrics, the metrics collector contains a mapping from the metric collected onto a function handler. The further inner-workings of metrics collector components are presented in Section 4.4.3. Instructions towards adding new metrics are summarized in Section 4.5.2.

To avoid mandating framework users to supply own user activity and virtual environment data, TheStick introduces (B) workload generator and (F) persistent data setup utilities. Workload generator, using a set of user-provided parameters, can generate a synthetic trace file that later can be used by benchmark executor. Persistent data setup utilities performs a similar action but with a persistent data - given user parameters it in some way interacts with MLG instance for example setting up player spawn points or spawning entities necessary to run a particular workload.

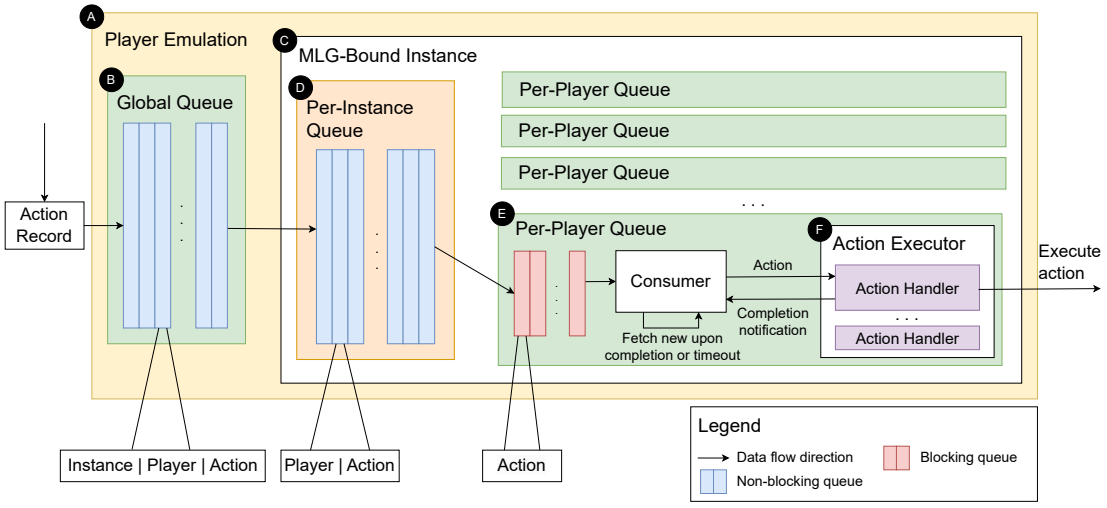


Figure 8: Player Emulation Design Overview

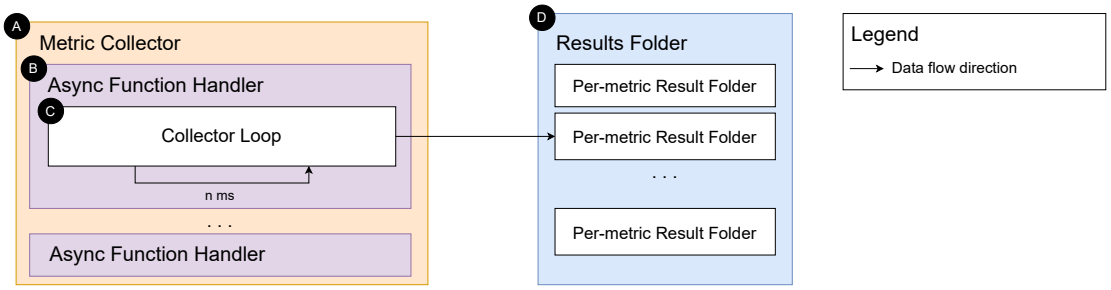


Figure 9: Player Emulation Design Overview

4.4.2 Player Emulation Design

To ensure action requirement of *atomicity* (**FAR1**) and ensure benchmark requirement of *generalization* (**FAR5**), we propose a player emulation component structure summarized in fig. 8. This is the only component in benchmark execution path which introduces a separation for different MLG modules. The separation is necessary as every MLG has its own set of allowed actions and their own protocols to executing the action.

The internal complexity of different modules for different MLGs is hidden behind the abstraction of B global queue which producer is benchmark executor from previous section. Queue entries are consumed and, based on information of which MLG the queue belongs to, messages are sent into D MLG specific queues. As soon as it is received, they are redirected into E per-player queues. Per-player queues have a F consumer attached that passes actions onto a particular F action handler responsible for executing the function. Upon receiving a completion notification, consumer fetches a new action. In case of timeout defined individually per action, consumer stops actions and then fetches a new one. This way, player emulator maintains requirement **FAR1** of *atomicity*.

4.4.3 Metrics Collector Design

4.5 Customizing Storage-Stick for Your MLG Server

4.5.1 Describing Specific to Your MLG Behavior: Adding Your Own Action

To every action added

4.5.2 From an Action to a Customized Workload: Composing Own Workload

To add your own action you have to update mapping from action name onto action handler in action handler files...

4.5.3 Recording Relevant to You Metrics: Adding Your Own Metric

4.5.4 Adding Your Own MLG

5 Modeling MLG User Storage Interaction

This section models MLG user-storage interaction answering **RQ1**. We start answering **RQ1** by, first, modeling the kind of data user can interact with in Section 5.1. Secondly, we model how MLG translates user packets into storage requests including possible performance optimization techniques in Section 5.2. Then, we model user behavior through minimal MLG user interactions - actions - that we define in Section 5.3. Finally, we combine all in MLG user storage interaction model Section 5.4 which includes both model construction in **??** and validation against Minecraft 1.2.0 - the latest version of biggest MLG representative at the time of the beginning of the project - in **??**.

5.1 Defining What Data Minecraft-Like Games Store: Data Model

This section models data required to be stored by MLG to maintain its persistence and modifiability properties. When constructing a model, we primarily base ourselves upon implementations of three biggest MLGs: Minecraft, Terraria, and Roblox. From those, we select common attributes and generalize them to exclude implementation-specific data attributes. We summarize the MLG model in fig. 10.

5.1.1 Terrain Data

We model all terrain to be made of up of some minimal units that make up the virtual world. Players can optionally interact with those minimal world units by placing, removing them, or changing their properties. All three selected games - Minecraft, Terraria, and Roblox implement their own minimal world units. In Minecraft's and Terraria's case it is called a block [6] [18], and Roblox's it is called a Part [13]. Following Minecraft's convention, the paper further refers to a minimal world unit s to a block. Each block is summarized to have at least *i) properties that distinguish that block from other blocks* and *ii) position that describes where the world is located on the global map. i)* Properties can be as simple as block id in Minecraft's and Terraria's case or more versatile color, size, mass, etc in case of Roblox's Part. Properties can be both static and dynamic - they can stay either unchanged throughout block's lifetime or they can be modified via players interaction. An example of dynamic property, in Minecraft's case, can be activated or disactivated state of redstone blocks or, in Roblox's case, any property can be set up as dynamic by using scripts [13]. We argue that *i)* properties and *ii)* position is a minimal set of attributes required to describe a block in MLG. Without *i)* properties, block is indistinguishable from one another and it become impossible to record results of user interaction with the block violating persistency property of Minecraft world. Without *ii)* position, it is impossible to know where the block is located and thus it would be impossible to record any block placement or

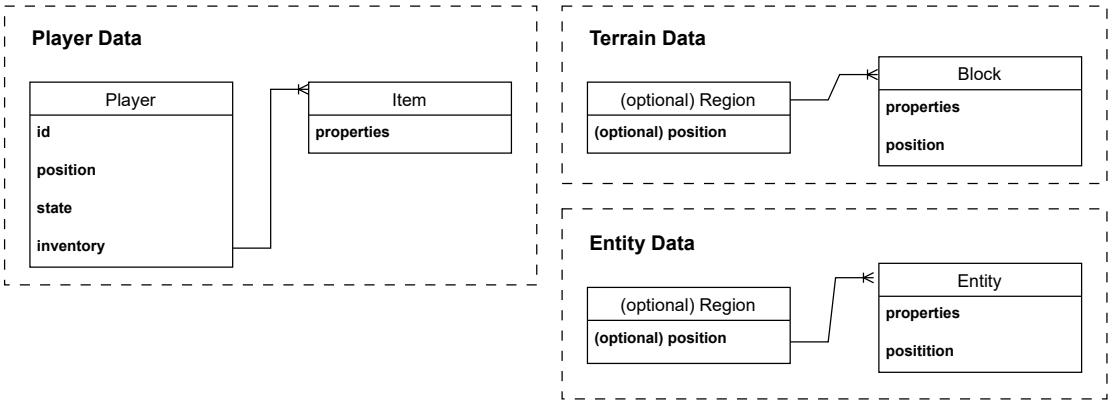


Figure 10: Minecraft Data Model Summary

removal violating a persistence property. To account for the case where the world is too big to fit in a memory at once - such as in Minecraft's or Terraria's case - we optionally group blocks in regions. Regions are present both in Minecraft [4] and in Terraria under the name of biomes [14].

5.1.2 Non-Playable Character Data

Non-playable characters (NPCs) are characters whose behavior is not controlled by a player. NPCs are present in all Roblox [12], Minecraft under the name entity [2], and Terraria under the name enemy [15] or NPC [17]. NPCs are internally organized similarly to terrain and have an *i) associated position* and *ii) properties that distinguish that NPC from others*. Alike blocks, *i)* properties can be both static and dynamic but, unlike for blocks, in NPC's case, dynamic properties can be changed both via player interaction with NPC or via internal game mechanics. We argue that properties and positions is a minimal set of data attributes required for NPC as without *i)* properties there is no way to record entity modification via player interaction or gaming mechanics and without *ii)* position there is no way there is no way to save the result of NPC movement. Absence of either would violate MLG's persistence property. Similarly to terrain data, to account for the case where the world is too big to fit in memory at once, we group entities based on the region they are currently in - technique adopted btoh in Minecraft and Terraria.

5.1.3 Player Data

We model every player having an associated player data entry describing their *i) id*, *ii) position*, *iii) state*, and *iv) inventory*. We define inventory as a set of *items* in player's possession. *Item* can be *blocks* that can be placed by a player or any other in-game items that can be used by a player to interact with MLG. All Roblox, Minecraft, and Terraria have their version of inventory [5] [11] [16]. Each item in player's inventory is distinguished from other items via their properties. We argue that for inventory and player data presented set of data attributes is minimal: without *i)* there would be no way to uniquely identify a player from others and uniquely link player's inventory to a particular player, without *ii)* there would be no way to record a result of player position change, and without *iii)* it would be impossible to record a result of in-game interactions with player character. Absence of any violates MLG persistence property.

5.2 Defining How Minecraft-like Games Store: Storage Process Model

In this section we model how MLG translates user interactions into storage requests along with optimization techniques MLGs can use to minimize storage performance impact on player's QoS. Model is summarized in fig. 11. After three biggest MLGs - Terraria, Minecraft, and Roblox - our model follows a client-server networking architecture.

We start by defining a (B) client and a (C) server. Client is controlled via (A) player via some form of input which can be mouse, keyboard, touchscreen or any others and in turn it it outputs a virtual world through some form of output that can be a screen or any other. Throughout the session, client constantly sends packets containing information regarding player's interaction. For example packet can have a semantic meaning of *player joining the world*, *player attacking another player*, *player changing position* or any other. Inside a server, there exists a (D) main thread with (E) game loop. The game loop with period defined individually per MLG discretely updates world state based on player's packets and internal game logic [29] [22]. Whenever player's packets processing requires interaction with a storage I/O *main thread* offloads the read to a (H) background thread to avoid stalling the game loop. Inside the *background thread* read operations are attempted to be served from game in-memory data contain *active regions* - regions with players currently playing - and *game cache* - generic in-game cache contents of which can be defined individually per MLG. Whenever request cannot be served from in-memory data - for example player spawns in non yet loaded region - the data will be requested from (I) file system. Serving some requests out of *in-memory* data allows to speed up storage I/O execution by serving some of them out of memory and avoid system calls and associated performance drop. The write operations are all written into (G) update buffer which every n seconds are flushed onto the storage. This allows to avoid a file system delay on each write operation and batch system write calls lowering the performance impact associated. Every MLG can define multiple update buffers with periods starting from 0 meaning that the data will be written straight into storage.

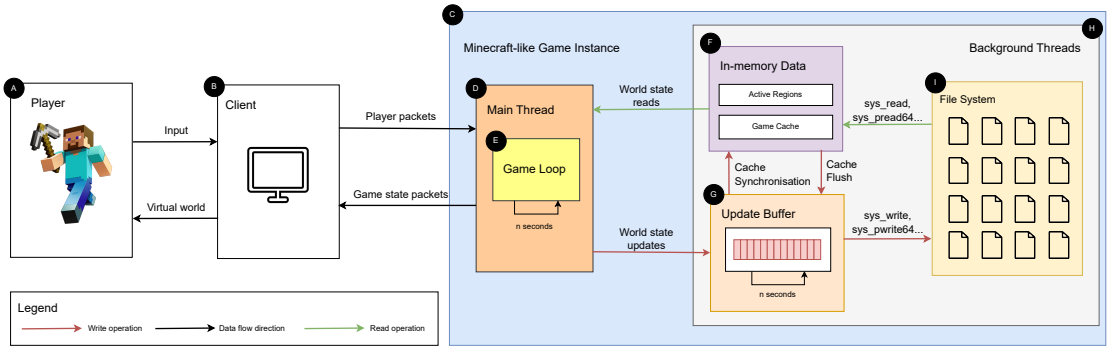


Figure 11: Minecraft-Like Games Storage Process Model.

The flush frequency would depend on tolerable level of inconsitency between in-memory data and the data persistently stored through the file system.

5.3 Actions - Instrument for Modeling User Interactions

This section defines models user interactions through *MLG action* - a core concept for user storage interaction model Section 5.4. We later, as well, reuse this concept for StorageStick workload customization **??**. We start by defining goals that we are trying to achieve with actions in Section 5.3.1. Then, based on goals, we constitute what defines a good list of actions for our purposes in Section 5.3.2 and Section 5.3.3.

5.3.1 The Definition and Goals of MLG Actions

We define MLG action is an atomic user interaction. The idea behind MLG actions is to comprise a list of building blocks that can be composed into arbitrary complex and fully customized workload. The intuition is that, if we can model a user storage interaction for a list of the atomic actions, then any arbitrary complex and customized workload can be modeled as a sum of the actions. This notion is later reused in **??** allowing MLG operators to asses storage impact on customized and suited for their own server MLG workload. Thus, we design MLG actions with goals of them being *i) compossible into a larger workload* and *ii) applicable to an arbitrary MLG user-storage interaction model.* To address *i)* for the final action list we introduce of set of workload compatibility requirements in Section 5.3.2. To address *ii)* we introduce a list of action model requirements in Section 5.3.3.

5.3.2 Action List Requirements to Target Workload Composability

With these requirements we want to ensure that the final derived action list provides a sufficient cover to construct a model of user storage interaction of an arbitrary workload. For that, using MLG behavior model defined in, we construct two coverage requirements (CR):

CR1 *Auxiliary Interaction Coverage*: we want to ensure coverage of player interactions both not only during and but also around MLG play as defined in Section 3.2. For that, we require the final actions list to contain auxiliary interactions - interactions not necessarily connected to MLG play but essential for MLG functionality.

CR2 *Player Behavior Coverage*: we want to ensure a coverage various player behavior during MLG play. For that, we require final list to contain user actions for all MLG behaviors described in Section 3.2.

To answer requirement **CR1**, we model auxiliary interactions in **??**. To answer requirement **CR2**, we model interactions per each behavior defined in Section 5.4.1.

Table 1: Included and Excluded Actions

Action	Interacts with data	Generalizable	Player Triggered	Included
Auxiliary Actions				
Player join	✔	✔	✔	✔
Player leave	✘	✔	✔	✘
Player kicked out	✘	✔	✔	✔
Player disconnected	✘	✔	✘	✘
Building and Mining				
Placing a block	✔	✔	✔	✔
Removing a block	✔	✔	✔	✔
Changing block properties	✔	✔	✔	✔
Fighting				
Player by Player Attack	✔	✔	✔	✔
NPC by Player Attack	✔	✔	✔	✔
Player by NPC Attack	✔	✔	✘	✘
NPC by NPC Attack	✔	✔	✘	✘
Exploring				
Player position update	✔	✔	✔	✔

5.3.3 Action Requirements for Modeling

We construct this set of requirements to decide whether the action should by included in MLG user-storage interaction model. Our three model requirements (MR) are:

MR1 *Unbounded to Implementation Functionality*: every action selected should be derived from MLG behavior model Section 3.2 and thus not be a part of any MLG-specific functionality.

MR2 *Generic Data Interaction*: every action should be modeled to interact with fields presented in MLG data model Section 5.1. Action cannot be modeled to interact with implementation-specific data.

MR3 *Player-Triggered*: every action should be triggered directly by player as actions triggered by internal game logic will be triggered under different conditions and thus will not generalizable

5.4 From User Interaction to Storage: Modeling Storage Interaction Through Actions

This section, using definition of MLG actions from Section 5.3, models user storage interaction. The section is structured based on the types of interactions defined in MLG behavior model Section 3.2. The list of actions that were modeled is summarized in table 1.

5.4.1 Constructing a Model

We start model construction from auxillary actions answering a requirement **CR1**. In MLG behavior model (Section 3.2), auxiliary interaction are defined to *player joining* and *player leaving* the server. We model *player joining* through a singular action of *player join* triggered via player's decision to join the world. *player leaving*, on the other hand, can occur under circumstances of: voluntary leave, leaving as a result of being kicked out by server administrators, or leaving as a result of internal game logic. The latter can, for example, occur if MLG decides player's network connection is not satisfactory to continue the play. All the actions do not require require storage interaction with any data defined in Section 5.1 in general case violating requirement *MR3*, and, thus, we do not model them. Additionally *player disconnected* it clearly is not *player-triggered* violating action requirement for generality **GR3**. Whenever Ⓐ *player join* occurs, we model MLG to read an associated Ⓛ player data and retrieve their spawnpoint. Spawnpoint can be contained both within *state* or *position* of

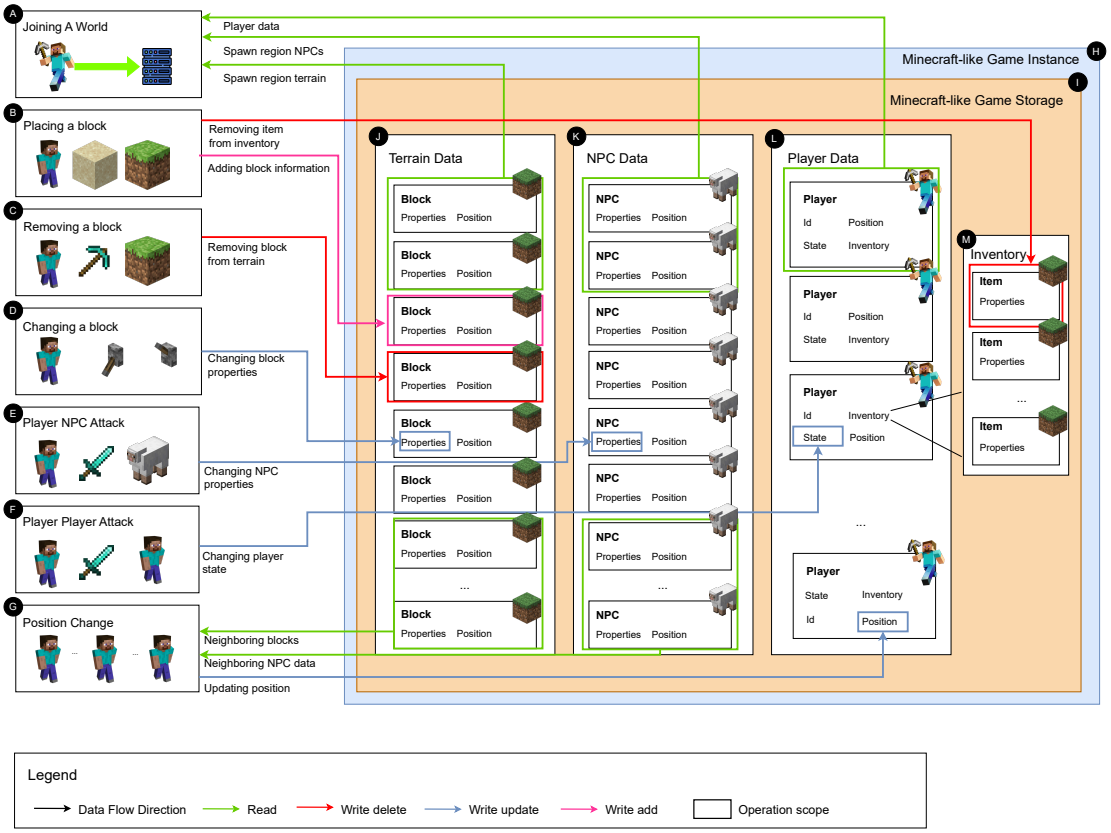


Figure 12: User Storage Interaction Model

player data (Section 5.1). Then, using the spawnpoint, MLG retrieves J terrain and K NPC data of region surrounding player's spawn.

6 Applying TheStick Framework to Study Storage Impact on Quality of Service - StorageStick Benchmark

This sections designs a StorageStick benchmark - an application of framework architecture defined in Section 4 to study an impact of storage on user QoS. As a result of this section, we answer **RQ2**. We begin a section by defining benchmark requirements in Section 6.1. Then, based on requirements, we define metrics and workloads in Section 6.2 and Section 6.3 and implement those on top of TheStick architecture.

6.1 Requirements

The final goal of StorageStick benchmark is to figure out an impact of storage onto Quality of Service. Quality of Service includes both average performance and performance guarantees. The performance guarantees are especially important for gaming workloads as lag spikes, a product of inconsistent performance, can significantly affect player's satisfaction. When analyzing how storage impacts both we take into account that storage request can be processed via multiple components each having orders of magnitude performance difference. This is why we want a benchmark to record both use and performance of different storage system components defined in Section 3.1. As well, we want our workloads to provide a comprehensive picture of storage impact onto QoS. To archive all, we define a following benchmark requirement (BR) list:

BR1 *Relevant Systems Metrics*: the benchmark should collect metrics of usage and performance of storage components defined via storage model in Section 3.1.

BR2 *Reflect Quality of Service Through Application Metrics*: application-level metrics collected via benchmark should reflect both the average performance experienced by user and a performance variability reflecting lag spikes that user can experience.

BR3 *Behavior coverage*: workloads should cover the whole range of behavior defined via behavior model in Section 3.2.

BR4 *Workload Parameter Coverage*: workloads should cover cases where different components, defined in Section 3.1, used to a different expect to reveal a comprehensive performance picture.

6.2 Metrics

This section defines system (Section 6.2.1) and application-level (Section 6.2.2) metrics that answer requirements defined in Section 6.1. We summarize a list in table 2.

6.2.1 System-Level Metrics

This section defines metrics corresponding to use and performance of every component defined in Section 3.1. We start by defining metrics for file system and then proceed to define metrics for storage medium.

File System Use and Performance

Number of File System Requests: this metric indicates file system use. This metric will indicate whether a workload is storage heavy or not. This is important when estimating whether workload performance can potentially be affected by storage. To get the metric, we process storage traces generated via Linux *strace* tool.

Read / Write Ratio: this metric indicates file system use. This metric will indicate whether a workload is read or write heavy. This is important when deciding on the underlying storage stack to use with regards to optimizing caching and reliability. To get the metric, we process storage traces generated via Linux *strace* tool.

Table 2: Benchmark Metrics Overview.

Metric	Level	Captures	Method
Number of file system requests	S	File system use	strace
Read / write ratio	S	File system use	strace
Indirect / direct requests ratio	S	File system use	strace
Cache hit / cache miss ratio	S	File system performance	perf
Median request latency	S	Storage medium performance	iostat
P99 request latency	S	Storage medium performance	iostat
Median queue depth	S	Storage medium use	iostat
P99 queue depth	S	Storage medium use	iostat
Median join latency	A	Average performance	timer
P99 join latency	A	Lag-spike performance	timer
Median tick time	A	Average performance	jfr
P99 tick time	A	Lag-spike performance	jfr

Indirect / Direct request ratio: this metric indicates file system use. This metric will indicate what is the maximum that can be served either out of read cache or to write buffer. This is important when deciding on file system caching policy and write size buffer. To get the metric, we process storage traces generated via Linux *strace* tool.

Cache Hit / Cache Miss Ratio: indicates file system performance - the higher the better. This metric will indicate how efficient is file system caching policy for a particular workload. Cache misses can have performance implication as their latency can be orders of magnitude higher than cache hits. To get the metric, we process data generated via *perf* tool.

Storage Medium Use and Performance

Median Request Latency: indicates storage medium performance - the lower the better. It is important as it can affect average application performance. To get the metric, we process data generated via *iostat* tool.

P99 Request Latency: indicates storage medium performance on its outlier - the lower the better. It is important as it can affect the application performance on the outliers. To get the metric, we process data generated via *iostat* tool.

Median Queue Depth: indicates storage medium use. It is important as it can be correlated with a higher median request latency and can be optmizes via increased number of queues - NVMe innovation. To get the metric, we process data generated via *iostat* tool.

P99 Queue Depth: indicates storage medium use. It is important as it can be correlated with a higher p99 request latency and can be optimizes via increased number of queues - NVMe innovation. To get the metric, we process data generated via *iostat* tool.

6.2.2 Application-Level Metrics

This section defined application metrics that showcase both average game performance and lag spikes. We analyze both as both are important for fulfilling an QoS service level agreement.

Average Performance Indicators

Median Join Latency: indicates how much time starting from sending an initial join request it takes to load up virtual environment data. This is an important metric as even slightly higher load time can affect user satisfaction. For example, Google reported 20% drop in traffic [23] when load time was increased by 0.5 seconds. To record this metric, we processed data generated via a timer incorporated in player emulation component.

Median Tick Time: indicates how long it takes to send out game state updates to players. Measured in both Yardstick and Meterstick [29] [22]. It is important as its value must not exceed a particular value - which in Minecraft case is defined as 50 ms; otherwise a server is considered to be overloaded. To record this metric, we process data generated via *jfr*.

Lag Spike Indicators

P99 Join Latency: indicates the amount of time to join a server in worst-case scenario. It is important as it indicates what QoS guarantees a server operator can provide. To record this metric, we processed data generated via a timer incorporated in player emulation component.

P99 Tick Time: indicates how long it takes to send out game state updates in a worst-case scenario. To record this metric, we process data generated via *jfr*.

6.2.3 Collection Methodology

6.3 Workloads

This section presents a list of pre-defined workloads that fulfills requirements **BR3** and **BR4**. We start from verifying storage impact on performance for workload consisting of auxilary action: *join the world* in Section 6.3.1. Then, we proceed to defining per-behavior workloads after behavior taxonomy presented in Section 3.2 in Section 6.3.2, Section 6.3.3, Section 6.3.4, Section 6.3.5, Section 6.3.6, Section 6.3.7.

6.3.1 Players Joining the World

6.3.2 Fighting: Attacking NPCs

6.3.3 Fighting: Attacking Other Players

6.3.4 Modifying Terrain: Building Blocks

6.3.5 Modifying Terrain: Removing Blocks

6.3.6 Modifying Terrain: Changing Redstone

6.3.7 Exploring The World

6.3.8 Workload 2

6.3.9 Workload 3

6.4 Systems

7 Evaluation

In this section, we apply a benchmark and pre-defined workloads defined in **??** on Minecraft 1.2.0 - the latest version of MLG at the time of starting a report. We start by introducing the main findings in Section 7.2 and then provide per-workloads findings in Section 7.3, Section 7.5. We provide experiment summary in table 3.

7.1 Experiment Storage Systems

7.2 Main Findings

7.3 Players Joining and Leaving the World

When executing experiments for the workload, we tested a system against all the storage configurations mentioned in Section 7.1. Then, we have defined parameters representing number of players joining, burstiness and preferential attachment. Each we have treated independently and for each we have defined three levels resulting in 27 conditions to test against. The workload total length for each was 60 seconds. For each condition we have collected 30 samples to tackle performance variability. We have summarized the all the variables in table 5.

7.4 Findings for Players Joining the World

7.5 Players Exploring

7.6 Players Fighting

7.7 Finding 2

7.8 Finding 3

Table 3: Experiments for players joining the world.

#	**Experiment Question**
	Players Joining and Leaving
1	How does storage latency affect time to join and tick time?
2	How does player preferential attachment defined in Section 3.4 affect time to join and tick time?
3	How does burstiness affect time to join and loop tick time?
4	How does number of joined players affect time to join and loop tick time?

Table 4: Latencies Used to Report Results.

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

Table 5: Players Joining and Leaving the World.

Level	**Preferential Attachment**	**Number of Player Joining**	**Burstiness**
Low	*Coefficient*: 0.7 *Stdev*: 100	5	1
Medium	*Coefficient*: 0.3 *Stdev*: 1000	10	2
High	*Coefficient*: 0.0 *Stdev*: 10000	25	5

References

[1] Nocom. URL https://2b2t.miraheze.org/wiki/Nocom.

[2] Entity format, . URL https://minecraft.wiki/w/Entity_format.

[3] Point of interest format, . URL https://minecraft.wiki/w/Point_of_Interest_format.

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

[5] Inventory – minecraft wiki, . URL https://minecraft.fandom.com/wiki/Inventory.

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

[7] Dimension – minecraft wiki, . URL https://minecraft.fandom.com/wiki/Dimension.

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

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

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

[11] Inventory | roblox wiki | fandom, . URL https://roblox.fandom.com/wiki/Inventory.

[12] NPC kit | documentation - roblox creator hub, . URL https://create.roblox.com/docs.

[13] Part | documentation - roblox creator hub, . URL https://create.roblox.com/docs.

[14] Biomes, . URL https://terraria.fandom.com/wiki/Biomes.

[15] Enemies - terraria wiki, . URL https://terraria.fandom.com/wiki/Enemies.

[16] Inventory, . URL https://terraria.fandom.com/wiki/Inventory.

[17] NPCs - terraria wiki, . URL https://terraria.fandom.com/wiki/NPCs.

[18] Blocks - terraria wiki, . URL https://terraria.fandom.com/wiki/Blocks.

[19] 2b2t. URL https://en.wikipedia.org/w/index.php?title=2b2t&oldid=1183700518. Page Version ID: 1183700518.

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

[21] 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.

[22] Jerrit Eickhoff, Jesse Donkervliet, and Alexandru Iosup. Meterstick: Benchmarking performance variability in cloud and self-hosted minecraft-like games. In *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering*, pages 173–185. ACM. ISBN 9798400700682. doi: 10.1145/3578244.3583724. URL https://dl.acm.org/doi/10.1145/3578244.3583724.

[23] Greg Linden. Geeking with greg: Marissa mayer at web 2.0. URL https://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html.

[24] 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.

[25] M. E. J. Newman. Clustering and preferential attachment in growing networks. 64(2):025102. ISSN 1063-651X, 1095-3787. doi: 10.1103/PhysRevE.64.025102. URL https://link.aps.org/doi/10.1103/PhysRevE.64.025102.

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

[27] 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/.

[28] Editorial Team. NVMe™ queues explained. URL https://blog.westerndigital.com/nvme-queues-explained/.

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